INTRODUZIONE AI **MICROCOMPUTER**

VOLUME 1

Il libro dei concetti fondamentali

di Adam Osborne

INTRODUZIONE AI
MICROCOMPUTER

VOLUME 1

Il libro
dei concetti
fondamentali

JACKSON ITALIANA EDITRICE
Piazzale Massari, 22 - 20125 Milano

SOMMARIO

SOMMARIO (continua)

CAPITOLO **PAGINA**

SOMMARIO (continua)

SOMMARIO (continua)

INDICE DELLE FIGURE

INDICE DELLE TABELLE

INDICE ANALITICO

INDICE ANALITICO (continua)

INDICE ANALITICO (continua)

INDICE ANALITICO (continua)

INDICE ANALITICO (continua)

INDICE ANALITICO (continua)

INDICE ANALITICO (continua)

INDICE ANALITICO (continua)

Figura 1—1. Chip e DIP di microcomputer.

Capitolo 1
CHE COS'E' UN MICROCOMPUTER

Un microcomputer è un dispositivo logico. Più precisamente, è una varietà indefinita di dispositivi logici, implementato su di un solo chip; ed a causa del microcomputer, la progettazione elettronica non sarà più la stessa.

La parola "microprocessore" è largamente usata anche in congiunzione con i microcomputer. Il termine "microprocessore" è stato coniato per dare l'idea delle funzioni limitate di questi dispositivi se paragonati ai computer, un microprocessore, quindi, rappresenta qualcosa di meno di un microcomputer. Attualmente si tende a far scomparire la distinzione fra "microprocessore" e "microcomputer"; perciò in questo libro useremo solo il termine "microcomputer", identificando con esso la logica implementata sui chip da una specifica funzione — usando la terminologia tradizionale.

CHIP

La Fig. 1-1 mostra un microcomputer. La logica del microcomputer è localizzata su un chip, montato in Dual In-Line Package (DIP).

DISPOSITIVO

Ci riferiamo al DIP come ad un dispositivo logico, opposto al sistema wafer, che è un chip logico. Il microcomputer è anche un computer digitale, come dice il nome stesso.

A dire il vero, vi sono forti somiglianze fra i microcomputer ed altri computer. Il metodo che viene usato per confrontare i computer — cioè per mezzo di set di istruzioni, dei metodi di indirizzamento e della velocità di esecuzione — fanno si che alcuni microcomputer assomiglino così tanto ad altri computer, che qualunque distinzione venga fatta fra i due prodotti, sembra fatta apposta solo per trovare una differenza.

Comunque i microcomputer sono un prodotto nuovo e diverso, e questa è la ragione per cui con essi non viene applicato il metodo che si usa per confrontare i computer. Il set di istruzioni, i metodi di indirizzamento e le velocità di esecuzione sono spesso di secondaria importanza per gli utenti dei microcomputer. Le caratteristiche di primaria importanza sono invece la distribuzione della logica sui chip e il prezzo dei microcomputer stessi: sono questi parametri che fanno dei microcomputer una cosa a parte rispetto a tutti gli altri computer, come prodotto nuovo e diverso.

Lo scopo di questo libro è non solo quello di spiegare cosa sono i microcomputer, ma anche chiarire perchè essi devono essere valutati molto diversamente rispetto a quanto si fa per gli altri computer.

Il libro non dà per scontato che voi sappiate già come opera un computer, perciò i vari concetti partono dai principi di base.

I microcomputer e tutti gli altri computer hanno comunque un antenato comune. Per acquisire una certa prospettiva, inizieremo quindi con una breve storia dell'evoluzione del computer e identificheremo le origini del microcomputer.

L'EVOLUZIONE DEI COMPUTER

Il microcomputer più piccolo del giorno d'oggi ed il più grosso computer a unità centrale, hanno un antenato in comune — l'UNIVAC 1, che è stato costruito con valvole nel 1950, e che occupava una stanza; ora esso avrebbe una capacità di elaborazione inferiore a quella della maggior parte degli attuali microcomputer.

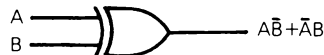
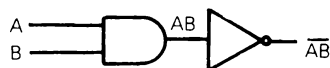
L'UNIVAC 1, ed i computer a valvole che lo seguirono, vennero usati per un numero molto limitato di applicazioni per cui non esistevano problemi di costo, spesso allo scopo di risolvere problemi matematici che altrimenti sarebbe stato impossibile affrontare.

La logica dei computer a valvole non era particolarmente adatta alle applicazioni scientifiche; tale logica era la conseguenza immediata e naturale del fatto di essere costruiti con dispositivi logici bistabili — il blocco fondamentale di ogni computer digitale.

A dire il vero, **i concetti di base per la progettazione di una macchina in grado di elaborare risalgono a Charles Babbage, che nel 1833 definì i concetti che possiamo trovare, con poche varianti, in tutti i computer digitali oggi costruiti. Nei Capitoli 2 e 3 descriviamo tali concetti base — concetti che permettono di derivare la logica del computer da cifre binarie, a prescindere da come il computer verrà usato.**

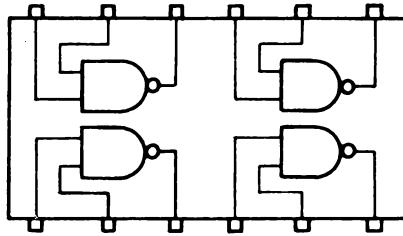
In pratica vogliamo dire che, da quando l'industria del computer è nata, non vi sono stati cambiamenti radicali nei concetti fondamentali dell'elaborazione. Sono stati i progressi nel campo della fisica a stato solido a rappresentare la forza che ha fatto evolvere l'industria del computer. La nuova tecnologia elettronica ha fatto diminuire i prezzi dei computer così rapidamente che, ogni pochi anni, nuovi mercati vengono invasi dei sistemi di elaborazione dati.

Nel 1960 i prezzi dei computer si abbassarono fino al punto da poter essere usati per il data processing, e arrivò così il giorno del computer general purpose.
Nel 1965 il PDP-8, al prezzo di 50.000$, portò i computer all'interno dei laboratori e delle linee di produzione industriali; nacque così l'industria del minicomputer. Oggi i minicomputer costano addirittura intorno ai 1.000$, e la loro sfera di influenza è ampliata nella misura in cui i prezzi sono diminuiti. Ma oggi i prezzi dei microcomputer vanno da $5 a $250 — e siamo così entrati in un'era in cui un computer può control-

lare una lavatrice o un forno, o può far parte di prodotti che rientrano nel consumo di massa.

Quali sono stati i progressi nel campo della fisica a stato solido di cui parliamo?

La valvola è un dispositivo voluminoso con elementi interni costosi. Negli ultimi cinquant'anni, è stata sostituita dal transistor, un piccolo pezzo di germanio, opportunamente drogato, come si dice in gergo tecnico.

Presto fu disponibile tutta una serie di componenti discreti, a basso costo.

In un chip vennero costruiti quattro gate NAND (dato che il prezzo era uguale, o di poco superiore, a quello di un solo gate NAND) per dare un buffer di tipo NAND positivo, quadruplo, a 2 ingressi.

Dispositivi di questo tipo produssero tutta una serie di altri dispositivi, ben noti ad una generazione di progettisti di logica, come i circuiti integrati serie 7400.

A dire il vero, i circuiti integrati della serie 7400, a loro tempo, diedero una forte spinta all'industria elettronica, come stanno facendo oggi i microcomputer; e questo perchè i circuiti integrati della serie 7400 convertirono tutta una generazione di "progettisti di circuiti" in una generazione di "progettisti di logica" — e tale cambiamento avvenne quasi dall'oggi al domani.

I quattro gate su di un chip divennero dieci, e poi cento, e poi mille; oggi su di un solo chip si possono implementare diecimila gate, e sicuramente siamo ancora ben lontani dal limite massimo.

CIRCUITI INTEGRATI SERIE 7400

Un chip sul quale vi è un certo numero di gate si chiama circuito integrato: se su di un chip vi sono circa da 100 a 1000 gate, facciamo riferimento alla logica come alla media scala di integrazione (o MSI). Ad un certo livello non definito, oltre i 1000 gate su di un chip, parliamo di Larga Scala di Integrazione (o LSI).

MEDIA SCALA DI INTEGRAZIONE

L'aspetto interessante dei circuiti integrati è che il costo di un chip è funzione della dimensione fisica — e non è della quantità di logica che è stata implementata sul chip. Perciò, man mano che i chip diventano più complessi, si possono costruire computer più economici.

LARGA SCALA DI INTEGRAZIONE

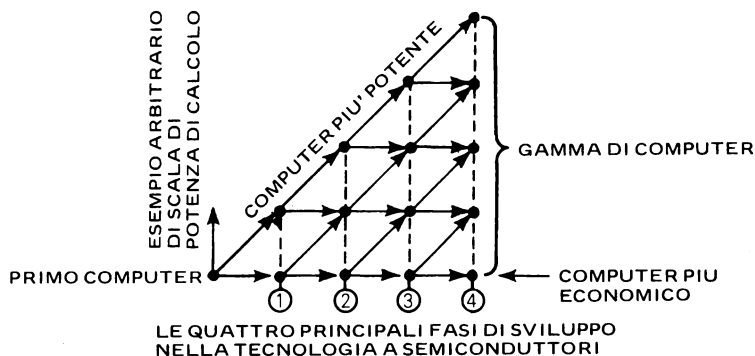
Vanno chiariti due aspetti circa il modo sorprendente in cui il computer si riduce in dimensioni:

1) Tutto il computer viene ridotto di dimensioni? E se no, quali parti rimangono le stesse?

2) Se il microcomputer è così poco costoso, perchè non ha eliminato tutti gli altri computer?

Prima di tutto, non si può ridurre di dimensioni tutto il computer, ma solo la parte elettronica. Ciò che rimane, è l'interfaccia uomo-computer — console e switch, mezzi per accettare l'ingresso dei dati e per generare risultati in una forma leggibile per l'uomo — tutte parti del computer che non sono necessarie se il computer diventa un dispositivo logico.

Il microcomputer non eliminerà mai tutti gli altri computer perchè quando i computer vengono usati per elaborare dati o risolvere problemi scientifici, vi è un'inflessibile necessità di rendere il computer più potente. Cosi, ad ogni progresso importante nella tecnologia dell'elettronica a stato solido, arrivate ad avere due nuovi prodotti: una versione più piccola del computer di ieri, ed una versione più potente del computer di oggi:

LE QUATTRO PRINCIPALI FASI DI SVILUPPO NELLA TECNOLOGIA A SEMICONDUTTORI

Col passare del tempo, vi è stato un notevole aumento delle capacità dei computer più economici rispetto a quelli più potenti. Così nel 1965 nasce la prima divisione arbitraria — fra i mini e i grossi computer. Non tenteremo di definire che cos'è un minicomputer in contrapposizione ad un grosso computer. Un minicomputer è un minicomputer perchè chi lo fabbrica lo chiama minicomputer.

Nel 1970, venne fatta una seconda divisione arbitraria, fra i minicomputer e microcomputer, ma questa volta le differenze fra i due prodotti sono più semplici da definire.

Un microcomputer viene venduto come uno, o pochi dispositivi logici, destinati a diventare componenti di un sistema logico più vasto.

Al contrario, tutti gli altri computer sono veicoli per l'esecuzione di programmi, ognuno dei quali definisce di volta in volta la funzione del computer.

Ma questa differenza fra un "minicomputer" e un "microcomputer" sta già scomparendo — per due ragioni:

L'HOBBY DEL COMPUTER

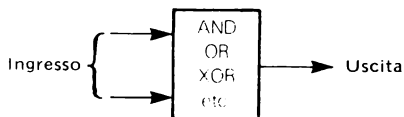
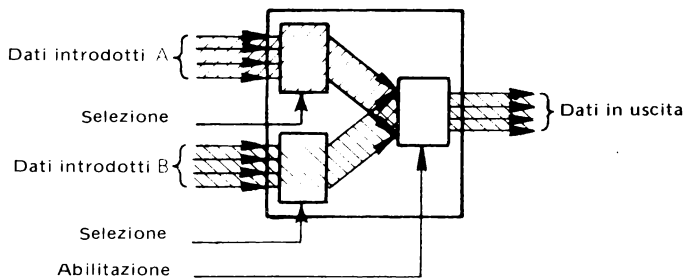
Primo, iniziano ad essere sempre più numerosi coloro che hanno l'hobby del computer: queste persone costruiscono il loro computer derivandolo da un microcomputer, scrivendone poi i programmi — proprio come farebbe qualunque programmatore di minicomputer. Secondo, un numero sempre maggiore di "microcomputer" non sono altro che l'implementazione in singolo chip di "minicomputer" già esistenti.

LE ORIGINI DEL MICROCOMPUTER

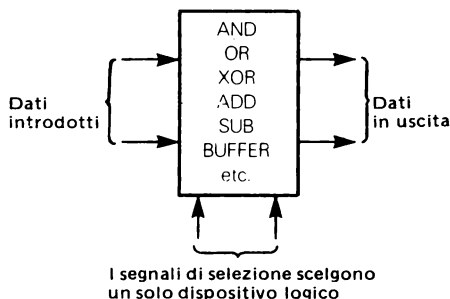
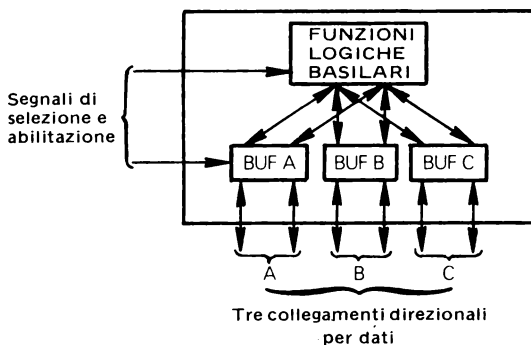
Dato che questo è un libro che tratta dei microcomputer, vediamo in che modo si è arrivati al primo vero microcomputer.

La Datapoint Corporation di San Antonio, Texas, fabbrica "terminali intelligenti" e piccoli sistemi di computer. Nel 1969, (insieme a Cogar e Viatron) tenta di fare un grosso salto in avanti. I tecnici della Datapoint progettano un computer molto semplice, e contrattano la Intel e la Texas Instruments per implementarlo su di un solo chip logico. La Intel vi riesce, ma il suo prodotto esegue le istruzioni circa dieci volte più lentamente di quanto la Datapoint aveva specificato; così la Datapoint rinuncia a comprarlo, e costruisce da sola il suo prodotto usando componenti dei logici già esistenti.

La Intel rimane con un dispositivo logico tipo computer, di cui era stato pagato lo sviluppo. Si trova quindi di fronte a due alternative: o fabbricarlo e venderlo, o lasciarlo su di uno scaffale. Sceglie di venderlo, lo chiama Intel 8008, e nasce così il primo microcomputer.

Nonostante il fatto che l'Intel 8008 fosse stato progettato per eseguire semplici elaborazioni di dati, il compito tradizionale dei computer esso creò un mercato che non era mai esistito prima: quello del dispositivo logico programmabile. Prendiamo in considerazione questo concetto. In qualunque catalogo di componenti logici, vi sono forse diecimila dispositivi logici diversi. Quelli più semplici li abbiamo già descritti; i semplici gate logici possono essere illustrati in questo modo:

Gli ingressi dati sono trasformati in uscite dati secondo i principi di alcune funzioni di trasferimento. Ma consideriamo un dispositivo logico più interessante: un buffer multiplexer a 4 bit e a due ingressi:

Vi sono due concetti interessanti in questi multiplexer. Primo, i dati vengono trattati in unità di 4 bit. Secondo, vi sono due segnali non-dati presenti: Select e Enable. Select determina quale ingresso dati diventerà l'uscita dei dati stessi. Enable determina quando si avrà l'uscita.

Se un chip LSI può contenere su di sè logica per centinaia di gate, per condensare un

intero catalogo di dispositivi logici su un solo chip universale, si può fare nel seguente modo:

Il chip universale ora illustrato contiene una grossa quantità di logica deplicata, non necessaria. Basandosi su poche funzioni logiche fondamentali, AND, OR, XOR, ADD, SUB – oltre che su pochi buffer e segnali di select e enable, si può sintetizzare uno qualunque dei diecimila chip elencati in un catalogo:

Il dispositivo logico di base può sintetizzare qualunque dispositivo logico individuale, o qualunque sequenza di dispositivi logici individuali.

Questo è il concetto di microcomputer.

A PROPOSITO DI QUESTO LIBRO

Lo scopo di questo libro è quello di farvi capire in modo completo che cosa sono i microcomputer e come si differenziano da altri computer. Dato che il libro non presuppone che abbiate già avuto contatti con i computer, i concetti base sono illustrati nei minimi particolari, e dai concetti base formiamo i componenti necessari di un sistema microcomputer.

Il libro intende sottolineare le differenze fra microcomputer e minicomputer.

Il libro non parla delle varie tecnologie che si usano per costruire i chip logici perchè, alla fine, la natura della tecnologia non è assolutamente importante per l'utente. Può darsi che le vostre applicazioni abbiano dei parametri chiave come il consumo o la velocità di esecuzione che potete tollerare; a dire il vero le varie tecnologie che si usa-

no influenzano certamente il consumo, la velocità di esecuzione ed altri fattori critici, ma dove questi fattori sono critici, il fatto di selezionare il computer giusto implica semplicemente il guardare le specifiche del prodotto. Il fatto di capire se il prodotto è fabbricato usando la tecnologia N-MOS o la tecnologia C-MOS, non rende particolarmente più difficile o più facile di capire che cos'è un microcomputer o come lo si usa.

COME E' STATO STAMPATO QUESTO LIBRO

Avrete notato che il testo in questo libro è stato stampato in neretto ed in chiaro. **Ciò è stato fatto per aiutarvi a saltare quelle parti del libro che coprono materia il cui argomento vi è familiare. Voi potrete essere sicuri che le scritte in chiaro sviluppano un'informazione presentata precedentemente in neretto.** Quindi leggete solo le scritte in neretto, finchè non troverete un argomento che voi volete conoscere maggiormente, a questo punto cominciate a leggere le scritte in chiaro.

Capitolo 2
ALCUNI CONCETTI FONDAMENTALI

Il motivo per cui vi sono differenze fondamentali fra un microcomputer e qualunque altro computer, è che tutta la produzione dei computer è basata sugli stessi concetti fondamentali di elaborazione — che ricadono tutti sotto un concetto logico fondamentale — quello della cifra binaria.

CIFRA BINARIA

Una cifra binaria è un numero che può assumere uno tra due valori: 0 o 1. Una cifra binaria non può avere alcun altro valore. Quello che rende la cifra binaria così utile è il fatto che essa può essere rappresentata da qualunque dispositivo bistabile. Qualunque cosa che può essere accesa o spenta, alta o bassa, può rappresentare uno zero in uno stato ed un uno nell'altro stato. La Fig. 2-1 illustra un dispositivo bistabile. E questa è tutta la fisica che avete bisogno di conoscere per capire i microcomputer.

Figura 2-1. Rappresentazione simbolica di cifre binarie secondo un dispositivo bistabile.

SISTEMI NUMERICI

Un computer che fosse in grado di contare fino a non più di uno non sarebbe una macchina molto utile. Fortunatamente, **le cifre binarie possono essere usate per rappresentare numeri di qualunque grandezza, come una stringa di cifre decimali può essere usata per rappresentare numeri superiori a nove.** Vediamo quindi in che cosa consistono realmente i numeri.

NUMERI DECIMALI

Quando un numero decimale ha più di una cifra, avete mai pensato a che cosa rappresenta realmente ogni cifra? Le due cifre 11 significano dieci più uno:

$$11 = 1\times10 + 1$$

In modo analogo, il numero 83 significa otto volte dieci più tre:

$$83 = 8\times10 + 3$$

Il numero 2347 significa due volte mille, più tre volte cento, più quattro volte dieci, più sette.

$$2\,3\,4\,7 = 2\times1000 + 3\times100 + 4\times10 + 7$$

Non c'è niente di singolare o di speciale a proposito dei numeri decimali. Il fatto che l'uomo abbia dieci dita nelle mani e dieci dita nei piedi è quasi sicuramente la ragione dell'uso universale dei numeri a base dieci, ma andrebbe ugualmente bene qualsiasi altra base.

NUMERI BINARI

Dato che le cifre decimali cessano di essere univoche con la cifra 9, il dieci deve essere rappresentato da "10", che significa una volta la base del numero (in questo caso, dieci) più 0. Usando la lettera "B" per rappresentare la base del numero, abbiamo:

$$10 = 1 \times B + 0$$

Ora nel sistema di numerazione binario, "B" non rappresenta dieci; rappresenta due. Perciò, **nel sistema binario, 10 = 2 decimale:**

$$10 = 1 \times 2 + 0$$

Analogamente, nel sistema binario, 11 rappresenta il tre decimale:

$$11 = 1 \times 2 + 1$$

Senza specificare, supponiamo che qualunque cifra del sistema numerico può essere rappresentata simbolicamente da d_i, d_j, d_k, ecc. Se B rappresenta la base del numero, qualunque numero può essere spiegato da questa equazione:

$$d_i d_j d_k d_1 = d_i \times B^3 + d_j \times B^2 + d_k \times B + d_1$$

Consideriamo un esempio decimale (B = 10) e un esempio binario (B = 2).

$$2\ 1\ 7\ 4 = 2 \times 10^3 + 1 \times 10^2 + 7 \times 10 + 4$$

$$d_i d_j d_k d_1 = d_i \times B^3 + d_j \times B^2 + d_k \times B + d_1$$

$$1\ 0\ 1\ 1 = 1 \times 2^3 + 0 \times 2^2 + 1 \times 2 + 1$$

CONVERSIONE DEI NUMERI DA UNA BASE ALL'ALTRA

CONVERSIONE DA BINARIO A DECIMALE

E' facile convertire i numeri da una base numerica ad un'altra. Dato che fin qui abbiamo parlato solo dei numeri decimali e binari, consideriamo la conversione dei numeri fra questi due sistemi.

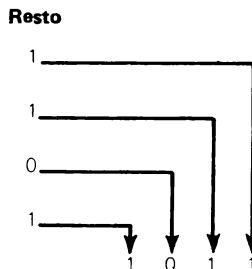
$$1\ 0\ 1\ 1 = 1 \times 2^3 + 0 \times 2^2 + 1 \times 2 + 1$$

$2^3 = 8$ e $2^2 = 4$, quindi

$$1\ 0\ 1\ 1 = 1 \times 8 + 0 \times 4 + 1 \times 2 + 1$$
$$= 8 + 0 + 2 + 1$$
$$= 11 \text{ (decimale)}$$

CONVERSIONE DA DECIMALE A BINARIO

La divisione continua per 2, tenendo conto del resto, fornisce un semplice metodo di conversione di un numero decimale nel suo equivalente binario; ad esempio, per convertire il decimale 11 nel suo equivalente binario, si procede nel modo seguente:

	Quoziente		Resto
$\frac{11}{2}$ =	5	+	1
$\frac{5}{2}$ =	2	+	1
$\frac{2}{2}$ =	1	+	0
$\frac{1}{2}$ =	0	+	1

Ne consegue che $11_{10} = 1011_2$

Gli indici 10 e 2 identificano i numeri come a base 10 e a base 2, rispettivamente.

CONVERSIONE DI FRAZIONI

L'equazione generale per convertire un numero binario frazionario nel suo equivalente decimale può essere scritta nel seguente modo:

$$d_i d_j d_k d_l \ldots etc. = (d_i \times B^{-1}) + (d_j \times B^{-2}) + (d_k \times B^{-3}) + (d_l \times B^{-4}) \ldots etc.$$

dove d_i, d_j, d_k rappresentano cifre numeriche e B rappresenta la base del numero.

Per esempio, per convertire 0, 1001_2 nel suo equivalente decimale, si procede nel seguente modo:

$$0{,}1011 = (1\times2^{-1}) + (0\times2^{-2}) + (1\times2^{-3}) + (1\times2^{-4})$$

dove $2^{-1} = \frac{1}{2^1} = 0{,}5$; $2^{-2} = \frac{1}{2^2} = 0{,}25$; $2^{-3} = \frac{1}{2^3} = 0{,}125$

$2^{-4} = \frac{1}{2^4} = 0{,}0625$

quindi $0{,}1011_2 = 0{,}5_{10} + 0 + 0{,}125_{10} + 0{,}0625_{10}$

$= 0{,}6875_{10}$

Per convertire un numero decimale frazionario nel suo equivalente binario (es. per convertire $0{,}6875_{10}$ nel suo equivalente binario), usate il seguente metodo di approssimazione:

Sfortunatamente, le conversioni di frazionari da binario a decimale non sono sempre esatte; nello stesso modo in cui una frazione come 2/3 non ha una rappresentazione decimale esatta, così una frazione decimale che non è la somma di 2^{-n} termini, si approssimerà soltanto ad una frazione binaria.

Consideriamo $0{,}42357_{10}$: la rappresentazione binaria di questo numero può essere eseguita nel modo seguente:

La risposta è $0{,}1101 \ldots_2$.

Per controllare, riconvertiamo:

$$0{,}01101 = 0\times2^{-1} + 1\times2^{-2} + 1\times2^{-3} + 0\times2^{-4} + 1\times2^{-5}$$
$$= 0 + 0.25 + 0{,}125 + 0 + 0{,}03125$$
$$= 0{,}40625_{10}$$

La differenza è 0,42357 – 0,40625, che è uguale a 0,01732; questa differenza è data dal resto trascurato, 0,55424. In altre parole, il resto trascurato (0,55424) mol-

tiplicato per il termine elaborato più piccolo (0,03125) dà l'errore totale:

$$0,55424 \times 0,03125 = 0.01732$$

ALTRI SISTEMI NUMERICI

Dato che i numeri binari tendono ad essere molto lunghi, le cifre binarie vengono spesso raggruppate in quelle di tre o quattro. I numeri hanno così o base 8 (ottale) o base 16 (esadecimale), come mostra la Tabella 2-1. Consideriamo il numero binario:

110111101100

NUMERI OTTALI

Raggruppando le cifre binarie in gruppi di tre, il numero viene convertito in formato ottale:

$$\underset{6}{110}\ \underset{7}{111}\ \underset{5}{101}\ \underset{4}{100} = 6754_8$$

La base 8 (ottale) comprende solo le cifre:

0, 1, 2, 3, 4, 5, 6, 7.

Il decimale 8 è equivalente all'ottale 10.

NUMERI ESADECIMALI

Raggruppando le cifre binarie in gruppi di quattro, il numero viene convertito nella base esadecimale:

$$\underset{D}{1101}\ \underset{E}{1110}\ \underset{C}{1100} = DEC_{16}$$

La base 16 (esadecimale) comprende le cifre:

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F.

Il decimale 16 è equivalente all'esadecimale 10.

Tabella 2-1. Sistemi di numerazione

ESADECIMALE	DECIMALE	OTTALE	BINARIO
0	0	0	0000
1	1	1	0001
2	2	2	0010
3	3	3	0011
4	4	4	0100
5	5	5	0101
6	6	6	0110
7	7	7	0111
8	8	10	1000
9	9	11	1001
A	10	12	1010
B	11	13	1011
C	12	14	1100
D	13	15	1101
E	14	16	1110
F	15	17	1111

ARITMETICA BINARIA

Sui numeri binari si può operare allo stesso modo che sui numeri decimali; infatti, l'aritmetica binaria è molto più semplice dell'aritmetica decimale. Prendiamo in considerazione l'addizione, la sottrazione, la moltiplicazione e la divisione binaria.

ADDIZIONE BINARIA

Le possibili combinazioni quando si sommano due cifre binarie sono:

Primo addendo	+	Secondo addendo	=	Risultato	+	Carry
0	+	0	=	0		
0	+	1	=	1		
1	+	0	=	1		
1	+	1	=	0	+	1

Il Carry, come nell'addizione decimale, viene aggiunto alla posizione binaria successiva più alta per esempio:

Questa addizione decimale è equivalente a questa addizione binaria:

Questa addizione decimale è equivalente a questa addizione binaria:

SOTTRAZIONE BINARIA

I microcomputer non possono sottrarre cifre binarie; possono solo sommarle. Fortunatamente non è un problema, dato che la sottrazione può essere convertita in addizione.

COMPLEMENTO A DIECI

Sottrarre un numero decimale equivale a sommare ad un numero il complemento a dieci.

Si trova il complemento a dieci di un numero sottraendo il numero stesso da 10. Il riporto finale, comunque, quando si eseguono sottrazioni decimali per mezzo dell'addizione del complemento di dieci, deve essere ignorato.

Consideriamo la sottrazione decimale

$$9 - 2 = 7$$

Il complemento a dieci di 2 è (10–2), che è uguale a 8. La sottrazione decimale può quindi essere eseguita per mezzo dell'addizione del complemento di dieci:

$$\begin{array}{r} 9 \\ +\ 8 \\ \hline =\ 1\,7 \end{array}$$

ignorare il carry finale

Eseguire la sottrazione decimale per mezzo della somma del complemento a dieci è sciocco, dato che 10–2 non è più semplice da calcolare di quanto non fosse 9–2. **L'equivalente binario di un complemento a dieci è un complemento a due.** Eseguire la sottrazione binaria per mezzo della somma del complemento a due ha invece molto più senso, inoltre, la logica del complemento a due si adatta ai computer.

COMPLEMENTO A UNO

Il complemento a due di un numero binario si ottiene sostituendo le cifre 0 con le cifre 1 le cifre 1 con le cifre 0, e poi aggiungendo 1. La prima fase genera un "complemento ad uno" di un numero binario. Per esempio, il complemento ad uno di 1011011101 è 0100100010.

Ecco alcuni esempi:

Numero binario	0101
Complemento ad uno	1010
Numero binario	1010100
Complemento ad uno	0101011

COMPLEMENTO A DUE

Il complemento a due di un numero binario si forma aggiungendo 1 al complemento ad uno di quel numero. Per esempio, il complemento ad uno di 0100 è 1011:

Numero originario	0100
Complemento ad uno	1011
	+1
Complemento a due	1100

Ora guardate come si può eseguire la sottrazione binaria aggiungendo il complemento di due del SOTTRAENDO al MINUENDO. Prima consideriamo la seguente sottrazione binaria

MINUENDO	10001
SOTTRAENDO	-01011
DIFFERENZA	00110

La stessa operazione può essere eseguita formando il complemento a due del sottraendo e aggiungendolo al minuendo. Il riporto finale deve essere scartato, proprio come doveva essere per la sottrazione del complemento a dieci:

Perciò la differenza è 00110.

Vediamo un altro esempio:

Quando si sottrae un numero più grande da un numero più piccolo, non vi è nessun

riporto da scartare. Consideriamo la versione decimale di questo caso. 2—9 diventa 2 + (10—9), o 2 + 1. La risposta, + 3, è il complemento a dieci del risultato negativo corretto, che è — (10 - 3) = -7. Ecco un esempio binario della stessa cosa:

101	Minuendo	101	Minuendo
-11011	Sottraendo	+00101	Complemento a due del sottraendo
-10110	Differenza	01010	La risposta negativa nella forma del complemento a due

Un numero binario più grande è stato sottratto da uno più piccolo. La risposta a destra è negativa, ma è nella forma del complemento a due; prendendo il complemento a due di 01010 (complemento a due = 10110), e dandogli il segno meno, si ottiene la stessa risposta di sinistra, —10110.

SEGNO DELLA RISPOSTA NELLA SOTTRAZIONE

Quando si esegue la sottrazione del complemento a due, il carry finale fornisce il segno della risposta. Se il Carry finale è 1, la risposta è positiva. (Il minuendo è maggiore del sottraendo), ed è nella sua forma positiva, quella del complemento a due.

MOLTIPLICAZIONE BINARIA

La moltiplicazione binaria è più facile della moltiplicazione decimale, dato che ogni prodotto parziale, in quella binaria, è o zero (moltiplicazione per 0) o esattamente il moltiplicando (moltiplicazione per 1). Per esempio:

Questa moltiplicazione decimale è equivalente a questa moltiplicazione binaria:

DIVISIONE BINARIA

La divisione binaria si può eseguire usando gli stessi passi della divisione decimale. Ecco un esempio:

ALGEBRA BOOLEANA E LOGICA DEL COMPUTER

L'algebra booleana è importante nelle applicazioni del microcomputer perchè essa fornisce le basi per la capacità decisionale, il test di condizioni e molte operazioni logiche.

L'algebra booleana usa le cifre binarie 0 e 1 per definire le decisioni logiche. Tre operatori booleani, OR, AND e OR esclusivo (XOR) creano una combinazione fra due cifre binarie per produrre un risultato ad una sola cifra. Un quarto operatore booleano, NOT, complementa una cifra binaria.

OPERAZIONE "OR"

L'operazione OR è definita, per due interi I e J, dalla seguente affermazione:

> **Se I o J, o entrambi, sono uguali a 1, il risultato è 1. Altrimenti il risultato è zero.**

Per rappresentare "OR", si usa un segno più (+). Dato che il simbolo booleano di OR si usa per rappresentare anche l'addizione aritmetica, non bisogna confondere i due operatori; essi sono molto simili, ma non identici. Su due cifre binarie si esegue una operazione di OR nel modo seguente:

$$0 + 0 = 0$$
$$0 + 1 = 1$$
$$1 + 0 = 1$$
$$1 + 1 = 1$$

Notate che l'ultima operazione OR $(1 + 1 = 1)$ è l'unica operazione OR in cui il risultato è diverso dall'addizione binaria.

TABELLE DELLA VERITA'

Le funzioni logiche si definiscono generalmente usando una Tabella della Verità che presenta l'elenco dei segnali di uscita associati con le combinazioni ammissibili dei segnali di ingresso. Segue la Tabella della Verità di un gate OR:

TABELLA DELLA VERITA' PER LA PORTA OR

INGRESSO		USCITA =
I	**J**	**I + J**
0	0	0
0	1	1
1	0	1
1	1	1

OPERAZIONE "AND"

L'operazione AND è definita , per due interi I e J, dalla seguente affermazione.

> **Se I e J sono entrambi 1, il risultato è 1. Altrimenti il risultato è 0.**

Il puntino · e il simbolo $\wedge$ vengono entrambi usati per rappresentare l'operazione AND. Le quattro combinazioni di 0 e 1 per l'operazione AND sono:

$$0 \cdot 0 = 0$$
$$0 \cdot 1 = 0$$
$$1 \cdot 0 = 0$$
$$1 \cdot 1 = 1$$

Segue la Tabella della Verità del gate AND:

TABELLA DELLA VERITA' PER LA PORTA AND

INGRESSO		USCITA =
I	**J**	**I • J**
0	0	0
0	1	0
1	0	0
1	1	1

OPERAZIONE "OR ESCLUSIVO"

L'OR esclusivo permette una differenziazione fra le cifre binarie d'ingresso che sono identiche e le cifre binarie in ingresso che sono diverse. L'uscita è 1 quando gli ingressi sono diversi e 0 quando gli ingressi sono uguali. Si usa il simbolo $\oplus$ o $\forall$ per rappresentare l'operazione XOR. Le quattro possibili combinazioni di 0 e 1 per l'operazione XOR sono:

$$0 \oplus 0 = 0$$
$$0 \oplus 1 = 1$$
$$1 \oplus 0 = 1$$
$$1 \oplus 1 = 0$$

Segue la Tabella della Verità di un gate XOR

TABELLA DELLA VERITA'
DI UNA PORTA OR ESCLUSIVO A DUE INPUT

INGRESSO		USCITA =
A	**B**	**A $\oplus$ B**
0	0	0
0	1	1
1	0	1
1	1	0

OPERAZIONE "NOT"

"NOT" complementa qualunque cifra o gruppo binario.

NOT 1 = 0
NOT 0 = 1

A causa della natura della logica del microcomputer NOT non è un'operazione logica particolarmente significativa; anzichè usare l'operazione NOT, si sfrutta la capacità del computer di generare un complemento ad uno.

La combinazione di AND con NOT genera NAND. La combinazione di OR con NOT genera NOR. I risultati di NAND e NOR sono il NOT di AND e di OR, rispettivamente.

Per rappresentare il NOT si mette un trattino sopra una cifra; perciò:

$$\bar{1} = 0$$
$$\bar{0} = 1$$

LE COMBINAZIONI DI OPERAZIONI LOGICHE

Un microcomputer non ha bisogno di avere tutti e tre gli operatori booleani AND, OR e OR esclusivo; si possono combinare alcuni operatori in modo da generarne altri, nel modo seguente:

A + B è riprodotto da $\overline{\bar{A} \cdot \bar{B}}$: ciò avviene nel modo seguente:

A	B	$A + B$	$\bar{A}$	$\bar{B}$	$\bar{A} \cdot \bar{B}$	$\overline{\bar{A} \cdot \bar{B}}$
0	0	0	1	1	1	0
0	1	1	1	0	0	1
1	0	1	0	1	0	1
1	1	1	0	0	0	1

L'OR esclusivo si può generare in questo modo:

$$A \oplus B = (A \cdot \bar{B}) + (\bar{A} \cdot B)$$

Ciò avviene nel modo seguente:

A	B	$A \oplus B$	$\bar{A}$	$\bar{B}$	$A \cdot \bar{B}$	$\bar{A} \cdot B$	$(A \cdot \bar{B}) + (\bar{A} \cdot B)$
0	0	0	1	1	0	0	0
0	1	1	1	0	0	1	1
1	0	1	0	1	1	0	1
1	1	0	0	0	0	0	0

TEOREMA DI DE MORGAN

Si possono combinare le operazioni booleane in modo tale da produrre una qualunque uscita desiderata da un set di ingressi conosciuti. Il teorema di De Morgan è un aiuto prezioso per la realizzazione di tali combinazioni. Il teorema può essere scritto in uno di questi due modi:

$$\overline{A \cdot B} = \bar{A} + \bar{B}$$
$$\overline{A + B} = \bar{A} \cdot \bar{B}$$

Perciò ad un microcomputer occorre solo un operatore booleano, OR, per generare tutti gli altri, **poichè**

$$A \cdot B = \overline{\bar{A} + \bar{B}}$$

genera AND da OR e NOT. Analogamente,

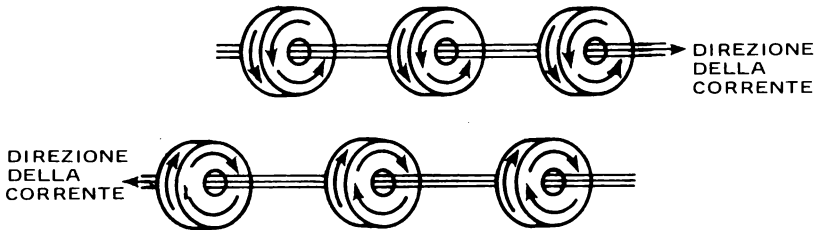
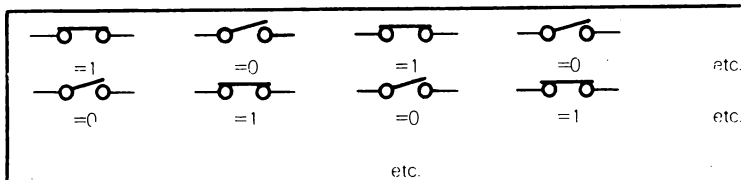
$$A \oplus B = \overline{(\bar{A} + B)} + \overline{(A + \bar{B})}$$

genera XOR da OR e NOT.

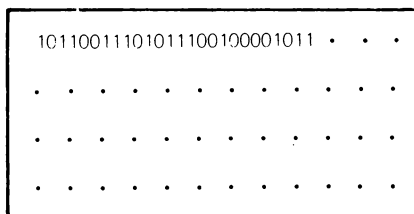
Capitolo 3
COME SI REALIZZA UN MICROCOMPUTER

BIT **Stabilito che le cifre binarie (Binary digit) (a cui si fa riferimento chiamandole bit) possono essere manipolate per eseguire una qualunque delle operazioni descritte nel Capitolo 2, come si combinano queste operazioni di base allo scopo di generare un microcomputer? Vediamo prima in che modo le informazioni vengono memorizzate in forma di dati binari.**

ORGANIZZAZIONE DELLA MEMORIA

I dati binari vengono immagazzinati nelle memorie. La memoria di un qualsiasi computer consiste di una Serie di elementi bistabili. Per i minicomputer si era soliti usare, e spesso si usano ancora, le memorie a nuclei, che consistono di piccoli toroidi metallici, che possono conservare una carica magnetica in senso orario o antiorario.

I microcomputer usano memoria a semiconduttore, che consistono di una serie di gate che possono essere conduttori e non conduttori:

MEMORIA NON VOLATILE **Le memorie a nuclei conservano la loro carica magnetica anche in mancanza di alimentazione;** potete estrarre una scheda di memoria a nuclei, inserirla in un altro computer simile al primo, e i dati in memoria dovrebbero essere ancora intatti. **Per questo motivo le memorie a nuclei sono chiamate "non volatili".**

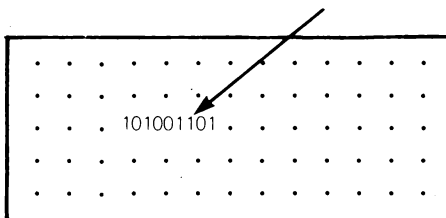
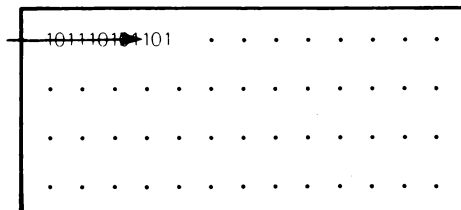
MEMORIA VOLATILE **Le memorie a semiconduttore perdono tutti i dati memorizzati nel momento in cui togliete l'alimentazione; perciò esse vengono chiamate "volatili".**

Non ha importanza il tipo di memoria usata in un microcomputer. E' necessario solo che la memoria consista di un numero di elementi bistabili, indirizzabili individualmente, ognuno dei quali rappresenta una sola cifra binaria:

Sono due le proprietà assolutamente necessarie che qualunque memoria deve avere:

1) **La posizione in cui viene memorizzata ogni cifra binaria deve essere indirizzabile in modo univoco.**
2) **Deve essere possibile leggere lo stato di ogni cifra binaria.**

ROM **Con alcune memorie non è possibile cambiare lo stato delle cifre binarie in memoria. Se lo stato delle cifre binarie può essere letto ma non cambiato, la memoria è chiamata una memoria a sola lettura, o ROM (Read Only Memory). Naturalmente, per la sua stessa natura, qualunque memoria ROM è non volatile.**

RAM **Se lo stato delle cifre binarie all'interno di una memoria può essere cambiato, nonchè venir letto, allora la memoria si chiama a lettura-scrittura. Generalmente si fa riferimento a tali memorie chiamandole memorie ad accesso casuale (RAM, Random Access Memory).**

Non c'è una precisa ragione per cui si dovrebbe riferire ad una memoria di lettura-scrittura, al contrario di quanto accade per una memoria di sola lettura, come ad una memoria accessibile casualmente; la memoria è accessibile casualmente se si può accedere direttamente alle cifre binarie individuali all'interno della memoria:

Se le cifre binarie all'interno di una memoria non fossero accessibili casualmente, sarebbero accessibili sequenzialmente, il che significa che alla decima cifra binaria, per esempio, si potrebbe accedere solo passando prima per tutte le cifre precedenti:

Le memorie di sola lettura-scrittura e le memorie di lettura-scrittura sono entrambe ac-

cessibili casualmente. Ciononostante, la terminologia comune fa riferimento alle memorie di sola lettura come alle ROM ed alle memorie di lettura-scrittura come alle RAM.

PAROLE DI MEMORIA

Il capitolo 2 spiegava come si combinano le cifre binarie per rappresentare i numeri superiori a 1, così come le cifre decimali si combinano per rappresentare numeri superiori a 9. La Tabella 2-1 dava alcune rappresentazioni binarie di numeri piccoli.

LUNGHEZZA DI PAROLA

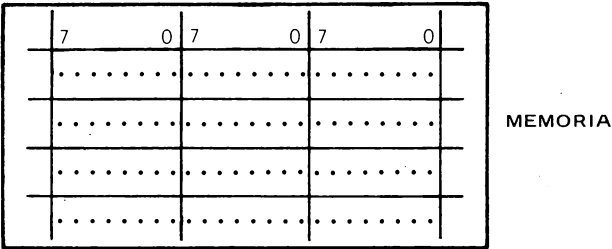
Il livello primario al quale le cifre binarie vengono raggruppate all'interno di qualunque computer è una delle caratteristiche più importanti del computer stesso, e si fa riferimento ad essa come alla lunghezza di parola del computer. Per esempio, un computer a 8 bit acquisisce tale definizione perchè i dati binari all'interno del computer saranno raggiunti ed elaborati in unità di otto cifre binarie. Una memoria organizzata in unità di 8 bit potrebbe essere visualizzata nel modo seguente:

Ogni puntino della figura precedente rappresenta una sola cifra binaria. Ogni riquadro rappresenta una parola a 8 bit.

Tabella 3-1. Lunghezza di parola dei computer

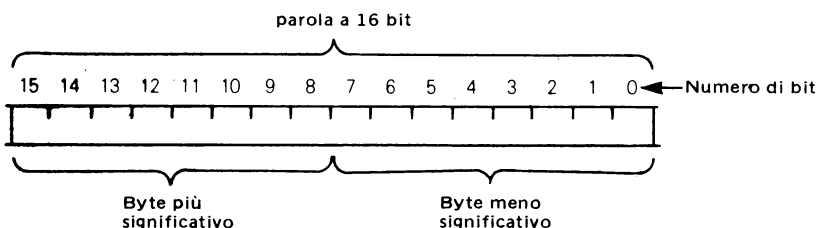
Dimensioni delle parole in bit	Microcomputer	Minicomputer	Grandi Computer
4	Molti	Nessuno	Nessuno
6	Nessuno	Pochi modelli obsoleti	Nessuno
8	I più comuni	Pochi	Nessuno
12	Pochi	Pochi	Nessuno
16	Pochi	I più comuni	Pochi
18	Nessuno	Pochi	Pochi
24	Nessuno	Pochi	Pochi
32	Nessuno	Pochi	I più comuni
64	Nessuno	Nessuno	Molto comune per i computer più grandi

Per convenzione i bit di una parola vengono numerati da destra (0 per il bit di ordine inferiore) verso sinistra (7 per il bit di ordine superiore), come mostra la figura. Alcuni costruttori di computer rovesciano la convenzione, numerando da sinistra a destra.

I costruttori di microcomputer, minicomputer e grossi computer hanno usato un gran numero di lunghezze di parole diverse. La Tabella 3-1 presenta un elenco delle lunghezze di parola più usate ed identifica quelle lunghezze di parola che vengono usate dai microcomputer, dai minicomputer e dai grossi computer.

La maggior parte dei microcomputer usano la parola di 8 bit. Vi sono alcuni computer a 4 bit, che tendono decisamente a rimpiazzare la logica digitale. Vi è anche un certo numero di microcomputer a 16 bit, che tendono invece a competere con i minicomputer sui loro mercati tradizionali.

IL BYTE

Un'unità di dati a 8 bit viene chiamata byte. Il byte è l'unità di dati più usata nell'industria dei computer; viene usata perfino dai computer che non hanno la parola di 8 bit. Un computer a 16 bit, per esempio, avrà spesso parole di memoria interpretate come due byte.

BYTE E PAROLE

Quando un microcomputer ha una parola di 8 bit, possiamo riferirci indifferentemente a "byte di memoria" e "parole di memoria": il significato è lo stesso.

Se la lunghezza della parola di un microcomputer non è di 8 bit, allora una parola di memoria e un byte di memoria non significano più la stessa cosa: un byte di memoria si riferisce ad un'unità di memoria di 8 bit, mentre una parola si riferisce ad un'unità di memoria della lunghezza della parola del microcomputer.

CIFRE

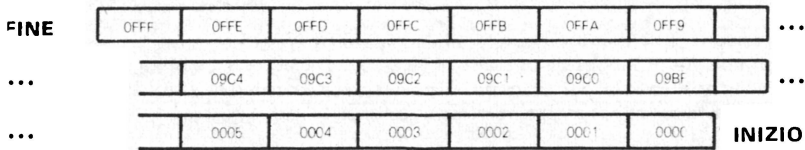
Molti microcomputer a 4 bit si riferiscono all'unità a 4 bit come ad una "cifra" (Nibble). Così ogni parola di memoria a 4 bit è una "cifra", e due parole di memoria a 4 bit costituiscono un byte.

INDIRIZZI DI MEMORIA

Anche se ogni cifra binaria all'interno di una memoria deve essere indirizzata in modo univoco, le cifre binarie non sono molto utili come entità singole; **perciò la più piccola unità di informazione a cui si accede solitamente in memoria è la parola.** Per esempio, quando si usa una memoria a 8 bit, ogni volta che si accede alla memoria, si fa riferimento ad otto cifre binarie.

Ogni parola di memoria ha un indirizzo di memoria univoco. Le parole all'interno della memoria hanno indirizzi di memoria sequenziali, con la prima parola in memoria che ha indirizzo, 0, e l'ultima che ha l'indirizzo più alto di qualunque altra parola di quella memoria. Il valore che l'indirizzo più alto avrà in quel momento dipenderà dalla grandezza della memoria. Perciò l'indirizzo di una parola è la sua posizione in

memoria; ad esempio, le parole di una memoria di 1000_{16} (4096_{10}) parole sarebbero indirizzate e numerate come segue (in notazione esadecimale):

FINE	0FFF	0FFE	0FFD	0FFC	0FFB	0FFA	0FF9		...
...		09C4	09C3	09C2	09C1	09C0	09BF		...
...		0005	0004	0003	0002	0001	0000	INIZIO	

Concettualmente vi sono delle sottili differenze fra il modo in cui i programmatori di minicomputer e quelli di microcomputer usano le memorie. Alcune di queste differenze vengono discusse adesso, mentre altre verranno descritte più avanti, poichè non avranno senso fino a che non verrà compresa come vengono usati i microcomputer.

CONCETTO DI MEMORIA NEL MINICOMPUTER

Per il programmatore di minicomputer, la memoria è semplicemente una sequenza di parole RAM indirizzabili individualmente, con indirizzi che iniziano a 0 e finiscono ad un dato numero elevato, che dipende dalla grandezza della memoria nel minicomputer. Avviene solo in rari casi che parte della memoria del microcomputer sia ROM. Sicuramente, **il programmatore di un minicomputer non deve mai preoccuparsi dell'implementazione fisica della memoria.** Fintanto che i dati possono essere memorizzati ed estratti su richiesta, non ha importanza dove e come questo succeda.

CONCETTO DI MEMORIA NEL MICROCOMPUTER

Al programmatore di microcomputer interesserà invece molto il modo in cui viene implementata la memoria, perchè vi sono molte applicazioni in cui un prodotto basato su di un microcomputer, una volta sviluppato, verrà venduto in decine di centinaia di unità. In un caso del genere, è molto importante che il numero dei componenti discreti all'interno di un microcomputer venga mantenuto al minimo dato che ogni componente extra (e quindi non necessario) verrà moltiplicato per decine di centinaia di volte — aumentado così i costi.

Il programmatore di microcomputer, inoltre, si interessa anche all'organizzazione della memoria perchè tutti i prodotti basati sui microcomputer usano delle ROM per una parte della memoria. La ragione per cui si preferisce l'uso della ROM nei microcomputer, è che la ROM è sicura.
Dato che all'interno della ROM non può essere modificata nessuna cifra binaria, nulla di ciò che è immagazzinato in tale memoria può venire accidentalmente cancellato.
L'utente di microcomputer considera la memoria come un insieme di chip di semiconduttori. La Fig. 3-1 illustra un dispositivo di memoria a 1024 bit in dual in-line package.

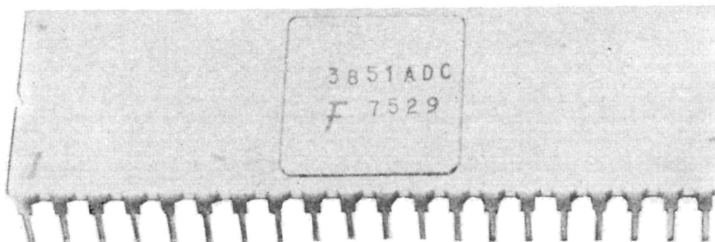
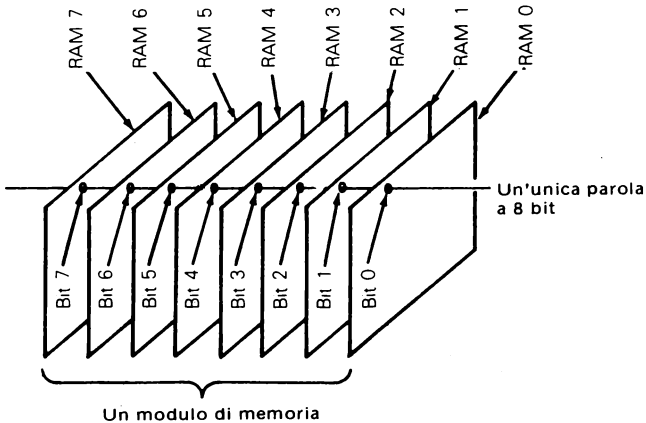
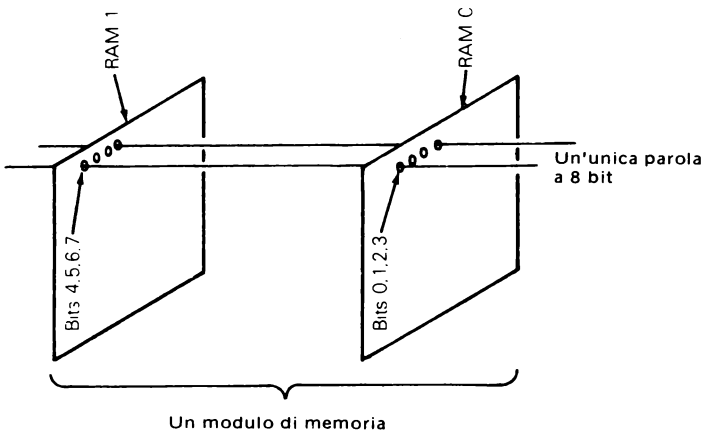
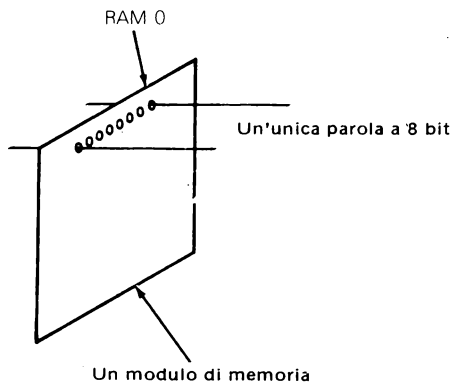



Figura 3-1. Dispositivo di memoria a 1024 bit

Solitamente la memoria ROM è implementata in singoli chip. Per esempio, un microcomputer può avere 1024 parole a 8 bit di memoria ROM su di un solo chip. Questo chip avrà una capacità di 8192 cifre binarie, divise in (e a cui si può accedere come a) 1024 unità di 8 bit. Al programmatore di un microcomputer interesserà come viene implementata la memoria perchè, per uscire dallo spazio di memoria fornito da un solo dispositivo ROM, bisogna aggiungere al sistema un dispositivo ROM addizionale.

La memoria di lettura-scrittura richiede più logica della memoria di sola lettura, dato che i bit individuali di una memoria di lettura-scrittura possono essere modificati oltre che letti. Perciò, la memoria di lettura-scrittura viene comunemente implementata su più di un chip. Prendendo ad esempio un caso molto semplice, otto chip RAM possono implementare parole di memoria di lettura-scrittura a 8 bit, ed ogni chip fornisce un bit della parola:

MODULO DI MEMORIA

Faremo riferimento a questo set di otto chip come ad un modulo di memoria. I piccoli sistemi microcomputer possono usare meno chip di memoria per implementare piccole memorie di lettura-scrittura. Per esempio, due chip RAM possono fornire ognuno quattro bit di una parola a 8 bit:

Un modulo di memoria

Ora nel modulo di memoria vi sono due chip di memoria.
Anche la memoria RAM è disponibile, come la ROM, con intere parole implementate su di un solo chip:

Il costo delle memorie RAM aumenta, in termini di costo unitario, per bit, quando, per implementare una sola parola di memoria si usano meno chip. Così l'implementazione di una memoria di lettura-scrittura a 8 bit usando otto chip RAM genera la memoria più economica. L'avere tutta la parola a 8 bit su di un chip RAM genera la memoria più costosa.

GRANDEZZA DI MEMORIA DEI CHIP RAM

Attualmente,sono di uso comune i chip RAM di 4096 bit, e stiamo per arrivare ai chip RAM a 16.384 bit. Entro, breve, dovrebbero essere messi a disposizione, in quantità commerciali, chip RAM a 65.536 bit — al prezzo di 5 — 10.000 lire per chip.

RAM DINAMICA

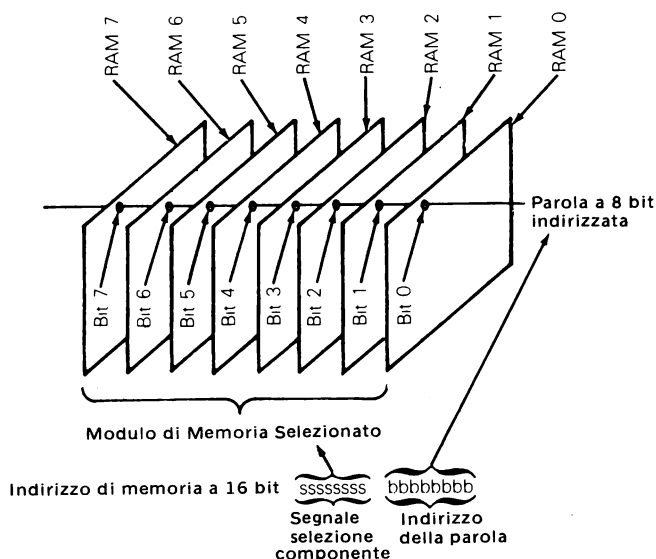
RAM STATICA

Vi sono due tipi di memoria RAM: RAM dinamica e RAM statica. La RAM dinamica, che è più economica, può solo conservare i dati per pochi millisecondi; perciò la RAM dinamica deve essere costantemente rinfrescata riscrivendo i contenuti delle parole di memoria. Il rinfrescamento della RAM dinamica viene fatto automaticamente in alcuni sistemi microcomputer; altri sistemi richiedono una logica di caricamento esterno nell'uso della RAM dinamica. La RAM statica costa di più, ma una volta che i dati vi sono stati scritti, vi rimangono fino a che viene fornita l'alimentazione.

Inoltre, come possibile utente di microcomputer, vi interesserà sapere esattamente quali indirizzi di memoria sono stati assegnati alla memoria di lettura-scrittura. Questo perchè gli indirizzi di memoria si traducono in chip RAM. Un byte in più di memoria dati può richiedere otto chip RAM nuovi, cosa che, moltiplicata per 10.000 diventa costosa.

Dato che, come utente di un microcomputer, vi preoccuparete costantemente del modo in cui i chip di memoria sono in relazione con gli indirizzi di memoria, questi andranno assumendo un significato che differisce notevolmente dal mondo dei piccoli e grossi computer. Più specificatamente, **ogni indirizzo di memoria può essere visualizzato sotto forma di bit che selezionano i chip e di bit d'indirizzo di parola.**

I bit di selezione del chip selezionano uno o più chip che costituiscono un modulo di memoria.
I bit d'indirizzo di parola identificano una parola di memoria all'interno del modulo di memoria selezionato.

Supponiamo che parole di memoria a 8 bit vengano implementate su otto chip di memoria separati. I bit di selezione del chip selezioneranno un modulo di memoria a otto chip. I bit d'indirizzo di memoria identificheranno una parola di memoria, nel modo seguente:

LUNGHEZZA DELL'INDIRIZZO DI MEMORIA

Il numero di bit d'indirizzo della parola richiesto da un modulo di memoria dipenderà dalla grandezza del chip. Per esempio, se un chip contiene una parte o tutte le 256_{10} parole di memoria, l'indirizzo della parola consisterà di otto cifre binarie:

Indirizzo di parola più piccolo $= 00000000 = 00_{16} = 00_{10}$

Indirizzo di parola più grosso $= 11111111 = FF_{16} = 255_{10}$

Un chip di memoria più grande può contenere una parte o tutte le 1024_{10} parole di memoria; l'indirizzo di parola consisterà allora di dieci cifre binarie:

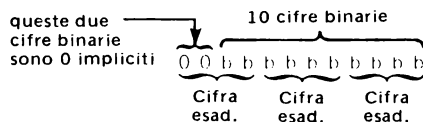
Indirizzo di parola più piccolo $= 0000000000 = 000_{16} = 000_{10}$

Indirizzo di parola più grosso $= 1111111111 = 3FF_{16} = 1023_{10}$

Notate che dieci cifre binarie creano tre cifre esadecimali nel modo seguente:

queste due cifre binarie sono 0 impliciti → **10 cifre binarie**

0 0 b b | b b b b | b b b b

Cifra esad. | **Cifra esad.** | **Cifra esad.**

LUNGHEZZA DELLA SELEZIONE DEL CHIP

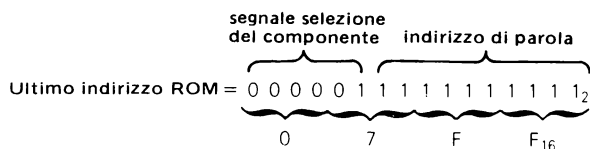
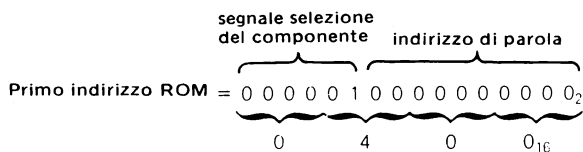
Il numero di bit di selezione del chip è funzione della architettura dei microcomputer; concatenando il numero di bit di selezione del chip con il numero di bit dell'indirizzo di memoria si ha la massima capacità di memoria del microcomputer. Per esempio, se il microcomputer può indirizzare 65.536_{10} ($FFFF_{16}$) parole di memoria, occorreranno 16 cifre binarie per esprimere il più alto indirizzo di memoria ammesso:

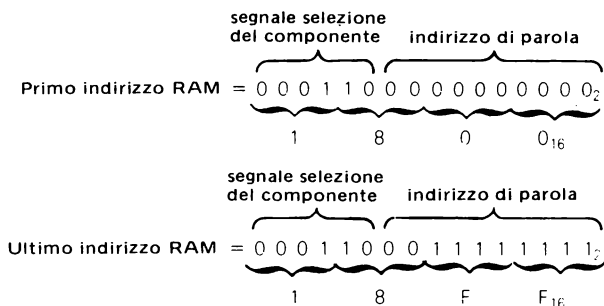
Ora se si usano chip di memoria di 1024 parole, l'indirizzo della parola necessita di dieci cifre binarie, lasciando quindi sei cifre binarie per la selezione del chip; in altre parole, la memoria massima consisterà di 64 moduli di memoria, con 1024_{10} parole per ogni modulo, e l'indirizzo di memoria a 16 bit deve essere interpretato nel modo seguente:

Quello che è importante ricordare è che il microcomputer usa il numero totale delle cifre binarie dell'indirizzo di memoria; come esse vengano divise fra la selezione del chip e l'indirizzo della parola di memoria dipende dal tipo di chip di memoria usati ed è completamente di competenza del progettista di logica.

Vediamo ancora come si forma in un caso reale un indirizzo di parola completo. Se, come mostrato in precedenza, l'indirizzo di parola all'interno del chip è 0110101110_2 ($1AE_{16}$) e la selezione del chip è 000111_2 (07_{16}), allora l'indirizzo di parola a 16 cifre binarie si forma nel modo seguente:

Non c'è nessun motivo per cui gli indirizzi di memoria disponibili debbano necessariamente essere contigui, o debbano necessariamente partire da zero. Per esempio, **un sistema microcomputer può comprendere un chip ROM implementando parole a 8 bit e un modulo RAM fatto di due chip RAM, ognuno dei quali implementa quattro bit su una parola a 8 bit.**

Se il chip ROM ha una selezione del chip di 000001_2 ed una capacità di 1024_{10} byte di memoria, gli indirizzi di memoria permessi andranno da 0400_{16} a $07FF_{16}$:

SPAZIO D'INDIRIZZO

Osservate che 1024_{10} byte di memoria avranno ora indirizzi tra 1024 e 2047_{10} o da 0400_{16} a $07FF_{16}$. Ci riferiamo a questa fascia di indirizzi di memoria chiamandola spazio d'indirizzo del modulo di memoria. Se il modulo RAM ha numero di selezione 000110_2 ed ogni chip contiene 256 x 4 bit, i due chip RAM costituiscono un modulo di memoria, e forniscono parole di memoria RAM a 8 bit con indirizzi da 1800_{16} a $18FF_{16}$.

Gli indirizzi che vanno da 1800_{16} a $18FF_{16}$ costituiscono lo spazio d'indirizzo del modulo RAM.

COME SI INTERPRETANO I CONTENUTI DELLE PAROLE DI MEMORIA

Una parola di memoria consiste di un certo numero di cifre binarie; perciò, le cifre binarie sono l'unica forma in cui l'informazione può essere memorizzata in una parola di memoria.

Una parola di memoria a 8 bit può contenere 256 (2^8) diverse configurazioni di zeri e di uni. Le configurazioni di zeri e di uni all'interno di una parola di memoria possono essere interpretate in uno dei due modi seguenti:

1) **Dati numerici binari puri a sè stanti.**
2) **Dati numerici binari che devono essere interpretati come parte di un'unità di dati a più parole.**

3) Un codice dati; cioè, una configurazione di bit soggetta a determinate interpretazioni arbitrarie predefinite.

4) Un codice istruzioni; cioè una configurazione di bit che deve essere trasmessa al microcomputer. Il microcomputer decodificherà la configurazione di bit e la interpreterà come identificativo di una delle operazioni che la logica del microcomputer deve immediatamente eseguire.

A questo punto l'unico concetto importante da capire è il seguente:

Basandosi sul solo esame dei contenuti di una qualunque parola, è impossibile determinare se essa contiene dati numerici, codici, o istruzioni.

Nel Capitolo 4 imparerete come un microcomputer si preoccupi del fatto che i contenuti di una qualunque parola di memoria possano essere interpretati in molti modi diversi. Ma prima descriveremo tutte le possibili interpretazioni di una parola di memoria.

DATI BINARI PURI

Consideriamo prima i dati binari puri, non soggetti cioè a speciali interpretazioni.

E' importante capire che potete rappresentare sulla carta dati binari puri, come un numero binario, un numero ottale, o un numero esadecimale, la scelta è solo questione di stabilire un modo di lettura anzichè un altro, e non influisce in alcun modo sulla parola dati. Ecco un esempio per una parola dati di 8 bit:

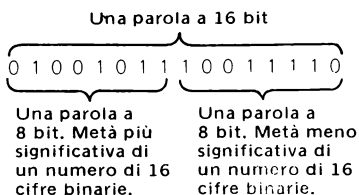
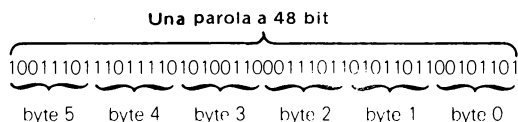
Ecco un esempio per una parola dati a 16 bit:

La scelta di una rappresentazione binaria, ottale o esadecimale per i contenuti di una parola di memoria non è interpretazione dei dati; è semplicemente un modo alternativo di scrivere la stessa cosa su di un pezzo di carta.

INTERPRETAZIONE DEI DATI BINARI

I contenuti di una parola di memoria, **interpretati come dati binari puri, possono essere definiti isolatamente, o parte di un'unità numerica più grande.**

DATI BINARI A PIU' PAROLE

Per esempio, una parola di memoria di 8 bit a sè stante può rappresentare valori numerici che vanno da 0 a 255_{10}. Una parola di memoria di 16 bit, invece, può rappresentare valori numerici che vanno da 0 a 65.535_{10}. **Non c'è ragione per cui parole di memoria di 8 bit non dovrebbero esser interpretate a coppie di 8 bit.** Ora i contenuti di ogni parola di memoria di 8 bit saranno interpretati come la parte meno significativa e la parte più significativa di un'unità di 16 cifre binarie:

Infatti non vi sono limiti al numero di parole di memoria che si possono concatenare per generare numeri molto grandi. Ecco un esempio di numero di 48 bit:

Supponiamo che per rappresentare una sola unità numerica siano necessarie sei parole di memoria di 8 bit, come illustrato prima. Normalmente, queste sei parole di memoria sono contigue, cioè hanno indirizzi di memoria fra di loro adiacenti. Però, non vi è nulla nella logica del microcomputer che richieda che i byte di un numero a più byte siano contigui. I numeri a più byte contigui sono più facili da elaborare; questo è l'unico motivo per cui si usa l'organizzazione contigua.

ADDIZIONE BINARIA DI PIU' BYTE

Che dire dell'aritmetica su più byte? I numeri che occupano molte parole di memoria possono essere addizionati, sottratti, moltiplicati o divisi usando le regole descritte nel Capitolo 2. In questo Capitolo nulla è stato detto circa il numero di cifre binarie associate ad un qualunque numero. Così se un numero di 16 bit è memorizzato in due parole di memoria di 8 bit adiacenti, si potrebbe usare ancora l'addizione binaria descritta nel Capitolo 2, ma i numeri di 16 bit andrebbero addizionati in due passi come segue:

Il riporto o carry, se c'è, di ogni passo è addizionato alla cifra di ordine inferiore del passo seguente.

Si evidenzia da sè l'estensione logica dell'esempio suddetto ai numeri memorizzati in tre, quattro o più parole di memoria. **Qui, ad esempio, vedete come due numeri, che occupano ognuno quattro parole di 8 bit verrebbero addizionati in quattri passi:**

	parola 3	parola 2	parola 1	parola 0
	10110100	10000101	01101011	11011010
+	01111010	10111010	01000010	00111001
1	1	0	1	
=	00101111	00111111	10101110	00010011
	passo 4	passo 3	passo 2	passo 1

SOTTRAZIONE BINARIA DI PIU' BYTE

C'è un trucco per sottrarre i numeri binari. Ricorderete dal Capitolo 2 che i dati binari vengono sottratti facendo il complemento a due del sottraendo e addizionandolo al minuendo, consideriamo due numeri di 16 bit memorizzati in parole di memoria di 16 bit. **La logica associata alla sottrazione di un numero di 16 bit da un altro, è molto semplice e può essere illustrata nel modo seguente:**

$$23A6_{16} - 124A_{16} = 115C_{16}$$

```
                          23A6₁₆ =  0010001110100110
Complemento a due di 124A₁₆ =  1110110110110101
                                                   1
                       Risposta =  0001000101011100
                                   1   1   5   C
```

Ora consideriamo la stessa sottrazione con i numeri memorizzati in due parole di memoria di 8 bit adiacenti. La sottrazione può essere riprodotta direttamente in questo modo:

Notate che solo il byte di ordine inferiore del numero viene complementato a due. Il byte di ordine superiore è complementato a uno. Anche se a prima vista ciò sembra creare confusione, non è così. Se voi visualizzate un numero a più byte come una singola unità numerica va da sè che quando si genera il complemento a due del numero a più byte, si aggiungerà 1 solo al byte di ordine inferiore a più byte:

NUMERI BINARI CON SEGNO

Un microcomputer che fosse in grado solo di elaborare numeri positivi non sarebbe molto utile. **Che dire a proposito dei numeri negativi? Qui, affrontiamo per la prima volta la questione dell'interpretazione di dati codificati in binario. Una convenzione industriale in vigore interpreta il bit di ordine superiore di un numero come bit di segno. Se questo bit è 1, il numero è negativo. Se questo bit è 0, il numero è positivo:**

0bbbbbbb rappresenta un numero positivo di 7 bit

1bbbbbbb rappresenta un numero negativo di 7 bit

La Tabella 3-2 dà le interpretazioni dei dati binari di 8 bit con segno. Osservate che i numeri negativi sono codificati come il complemento a due dei loro corrispondenti positivi. Ecco alcuni esempi:

$+02_{16}$ = 00000010 $\qquad$ -02_{16} = 11111101
$\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad$ 1
$\qquad\qquad\qquad\qquad\qquad\qquad$ 11111110

$+6A_{16}$ = 01101010 $\qquad$ $-6A_{16}$ = 10010101
$\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad$ 1
$\qquad\qquad\qquad\qquad\qquad\qquad$ 10010110

Quando otto cifre binarie vengono interpretate come numero con segno, il campo di numeri rappresentabili va da -128_{10} a $+127_{10}$. Quando sedici cifre binarie sono interpretate come un numero binario con segno, i numeri sono compresi tra da -32768_{10} a $+32767_{10}$.

Tabella 3-2. Interpretazioni numeriche binarie

BINARIE	DECIMALI EQUIVALENTI	ESADECIMALI
10000000	-128	80
10000001	-127	81
10000010	-126	82
10000011	-125	83
⋮	⋮	⋮
11111110	-2	FE
11111111	-1	FF
00000000	0	0
00000001	1	1
00000010	2	2
00000011	3	3
⋮	⋮	⋮
01111101	+125	7D
01111110	+126	7E
01111111	+127	7F

Il vantaggio di usare questo metodo per rappresentare numeri con segno sta nel fatto che esso non richiede una logica speciale quando si eseguono le operazioni aritmetiche. L'eseguire una operazione aritmetica non genera una risposta che è troppo grande per essere contenuta nello spazio disponibile, quindi si può ignorare il segno di un numero finchè non se ne voglia interpretare il risultato; a questo punto, l'esame del bit di ordine superiore del risultato indica se esso è positivo o negativo. Se il bit di ordine superiore del risultato è 1, allora il numero è negativo, e prendendone il complemento a due ne ottiene la rappresentazione binaria positiva. Ecco alcuni esempi:

Considerate ancora questo esempio, riscritto nel seguente modo $(3A_{16}) + (-63_{16}) = (-29_{16})$. (-63_{16}) sarà rappresentato dal complemento a due di $+63_{16}$.

$+63_{16} = 01100011$
complemento a uno di $+63_{16} = 10011100$
$63_{16} = 10011101$

Quindi:

$3A_{16}$ 00111010
$+(-63_{16})$ 10011101
$=(-29_{16})$ 11010111

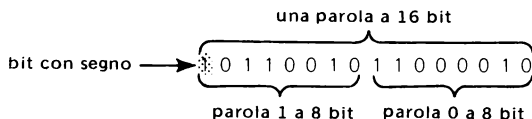
Questo uno indica un risultato negativo

00101000
1
00101001
2 9

Prendendo il complemento a due del risultato si ottiene una rappresentazione positiva.

Osservate che usando il complemento a due per rappresentare numeri binari con segno, $3A_{16}$ -63_{16} e $3A_{16}$ + (-63_{16}) hanno rappresentazioni binarie identiche, cosa che c'era da aspettarsi per via dello schema di rappresentazione di numeri negativi.

NUMERI BINARI A PIU' PAROLE CON SEGNO

Tali numeri non presentano problemi particolari se è chiaro che le operazioni devono essere eseguite una parola alla volta. Lo vedete illustrato qui di seguito, per il semplice caso di dati binari di 16 bit con segno, che generano gli stessi risultati sia che vengano trattati come una sola parola di 16 bit, o due parole di 8 bit.

Prendiamo in considerazione la sottrazione di due numeri binari di 16 bit con segno, dove ogni numero è memorizzato in due parole di 8 bit. Come l'addizione a più parole senza segno, anche l'addizione a più parole con segno viene eseguita in due passi, in questo modo:

Osservate che -0810_{16} viene generato prendendo il complemento a due di 0810_{16}, come segue:

0810_{16} = 0000100000010000
complemento a uno = 1111011111101111
complemento a due = 1111011111110000

BINARIO DECIMALE CODIFICATO

E' possibile codificare numeri decimali usando cifre binarie. Quattro cifre binarie possono rappresentare valori da 0 a F_{16}, o da 0 a 15_{10}. Ignorando le combinazioni di cifre binarie al di sopra del 9, i numeri decimali possono essere codificati, due cifre per ogni parola di memoria di 8 bit, o quattro cifre per ogni parola di memoria di 16 bit. **La Tabella 3-3 identifica le combinazioni di quattro cifre binarie che possono essere interpretate come numeri decimali.** Quando le cifre binarie vengono usate per rappresentare numeri decimali, il risultato si chiama dato Binario Decimale Codificato (BCD, Binary Coded Decimai).

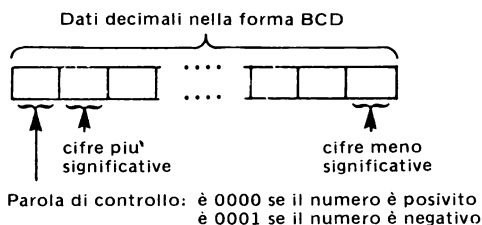
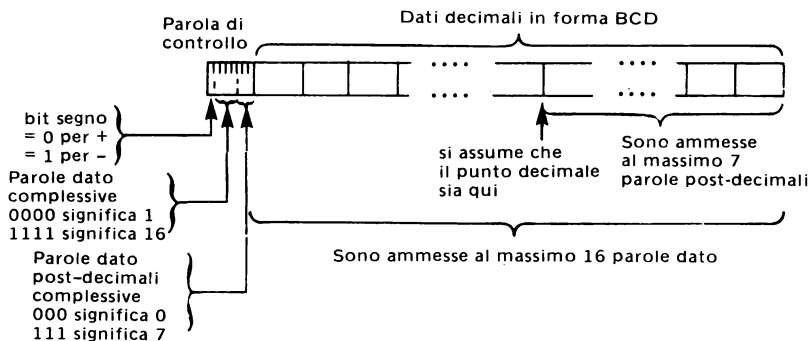
DATI BCD NEGATIVI

Le regole usate per i numeri binari con segno non possono essere applicate ai dati BCD, dato che il codice BCD richiede che i dati binari siano interpretati in unità di 4 bit:

Ogni cifra di 4 bit può avere una delle configurazioni di bit mostrate nella colonna BCD della Tabella 3-3. Una parola di 8 bit usa il bit di ordine superiore per tutti i numeri a 79_{10}. Se il bit di ordine superiore è necessario per rappresentare le cifre decimali 8 o 9, esso non può essere usato per rappresentare il segno.

Tabella 3-3. Rappresentazione binaria di Cifre Decimali

BINARIA	ESADECIMALE	BCD
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	A	Illegale
1011	B	Illegale
1100	C	Illegale
1101	D	Illegale
1110	E	Illegale
1111	F	Illegale

Il segno dei numeri BCD con segno viene quindi rappresentato usando una speciale parola di "controllo", che deve precedere la prima parola dati di un numero BCD a più parole. Non esistono regole generali per il formato della parola di controllo, ma vi diamo qui un esempio semplice ed uno complesso. Prima quello semplice:

Ecco ora l'esempio complesso:

I dati BCD non possono essere addizionati e sottratti usando le regole dell'addizione e della sottrazione binaria diretta. Ecco alcuni esempi degli errori che si potrebbero fare:

Decimale		BCD	Decimale		BCD
23	=	00100011	54	=	01010100
+47	=	01000111	−26	=	11011010
=70		01101010	=28		00101110
		6 **Illegale**			2 **Illegale**

Notate che 11011010 è il complemento a due della rappresentazione binaria di 26.

ARITMETICA BCD

Per eseguire operazioni aritmetiche su numeri codificati BCD, bisogna applicare regole speciali, e bisogna tener conto del Carry di ogni cifra BCD. Consideriamo l'addizione:

```
  97
 +68
=165
```

Non è sufficiente tener conto del fatto che c'è stato un carry nell'addizione della cifra di ordine superiore. Bisogna tener conto di qualunque carry intermedio nell'addizione della cifra di ordine inferiore. Ecco alcuni esempi che mostrano lo stato del carry (C) e del carry intermedio (IC):

C IC	C IC	C IC	C IC
↓ ↓	↓ ↓	↓ ↓	↓ ↓
0 0	0 1	1 0	1 1
21	29	91	97
+32	+32	+32	+68
=53	=61	=123	=165

Concettualmente, l'addizione BCD viene eseguita nel modo seguente:

1) Addizionare i due numeri
2) Addizionare in modo seriale 6 ad ogni cifra non valida, partendo dalla cifra di ordine inferiore. (Questo "saldo di sei" delle configurazioni binarie non valide viene eseguito per avere la rappresentazione BCD corretta).

```
Per esempio, 25₁₀+19₁₀ = 0010  0101
                         0001  1001
                         ----------
                         0011  1110
          aggiungendo 6: 0000  0110
                         ----------
                         0100  0100 = 44₁₀
```

Questo metodo è piuttosto tortuoso da elaborare. Il processo che segue, descritto per due cifre di un numero BCD, è più efficace; esso aggiunge 6 ad ogni cifra e poi basandosi sugli stati del carry, sottrae quei sei che non erano necessari:

1) Usando l'addizione binaria, aggiunge 66_{16} al primo numero (l'augendo).
2) Aggiunge il secondo numero (l'addendo) alla somma generata nella I° fase. Il carry generato in questa fase riflette il vero carry della cifra superiore seguente.

3) Usando l'addizione primaria, aggiunge un fattore alla somma 2° fase. Il fattore da aggiungere dipende dal carry (C) e dal Carry intermedio (IC), in questo modo:

C	**IC**	**Fattore**
0	0	$9A_{16}$
0	1	$A0_{16}$
1	0	FA_{16}
1	1	00_{16}

Ecco alcuni esempi:

	23 +32 =55	29 +34 =63	92 +32 =124	87 +79 =166
Primo addendo =	00100011	00101001	10010010	10000111
66_{16} =	01100110	01100110	01100110	01100110
Passo 1, somma =	10001001	10001111	11111000	11101101
Secondo addendo =	00110010	00110100	00110010	01111001
Passo 2, somma =	10111011	11000011	00101010	01100110
C/IC =	0 0	0 1	1 0	1 1
Fattore =	10011010	10100000	11111010	00000000
Passo 2, somma =	10111011	11000011	00101010	01100110
Risultato =	01010101	01100011	00100100	01100110
=	5 5	6 3	2 4	6 6
C (dal passo 2) =	0	0	1	1

Per un numero BCD di due cifre, la sottrazione BCD viene eseguita attraverso queste due fasi:

1) Aggiungere il complemento a due del sottraendo (il numero che viene sottratto) al minuendo (il numero dal quale si sottrae). Il carry generato in questa fase riflette il vero carry della cifra superiore che segue.
   Ricordate che quando si sottraggono numeri a più parole, solo la parola di ordine più basso del sottraendo viene complementata a due. Le parole di ordine superiore vengono complementate a uno.
2) Eseguite la terza fase, come descritta per l'addizione BCD.

Ecco alcuni esempi di sottrazione:

	75 −21 +54	71 −28 +43	25 −71 −46	21 −78 −57
Sottraendo =	00100001	00101000	01110001	01111000
Complemento a due =	11011111	11011000	10001111	10001000
Minuendo =	01110101	01110001	00100101	00100001
Passo 1, somma =	01010100	01001001	10110100	10101001
C/IC =	1 1	1 0	0 1	0 0
Fattore =	00000000	11111010	10100000	10011010
	01010100	01001001	10110100	10101001
Risultato =	01010100	01000011	01010100	01000011
=	5 4	4 3	5 4	4 3
C (dopo AC) =	1	1	0	0

Quando si eseguono sottrazioni BCD, viene indicato un risultato negativo da un carry finale di 0 (come per la sottrazione binaria) ma mantenendo la rappresentazione decimale dei numeri, il valore numerico della risposta negativa è nella forma del complemento a dieci, non nella forma del complemento a due. Così la risposta di 25 - 71 appare come 54, che è 100 - 46, e il riporto finale è 0. Analogamente, la risposta di 21 - 78 appare come 43 che è 100 - 57, e il riporto finale è 0.

CODICI CARATTERI

Un computer non sarebbe utile se si dovesse introdurvi i dati come una sequenza di cifre binarie, o se le risposte fossero fornite in uno dei formati binari codificati o non codificati. Deve essere possibile per un computer trattare testi ed altre informazioni non numeriche.

Se teniamo conto che la combinazione di cifre binarie all'interno di qualunque parola di memoria può essere riusata un numero qualunque di volte, tutti i codici binari che sono stati usati per rappresentare dati numerici, come descritto fin qui, possono essere riusati per rappresentare lettere dell'alfabeto, caratteri digitali, o qualunque altro carattere speciale.
Finchè un programma interpreta correttamente le cifre binarie di una parola di memoria, non possono sorgere confusioni e ambiguità.

Per esempio, se voi come programmatore decidete di usare parole di memoria con indirizzi da $0A20_{16}$ a $0A2A_{16}$ per mantenere i dati decimali codificati in forma binaria, allora siete voi, il programmatore, nella vostra logica successiva, a dover ricordare che i dati binari in queste parole di memoria devono essere interpretati come cifre decimali codificate in forma binaria — e che qualunque altra interpretazione darà luogo a degli errori.

Allo stesso modo, se le parole di memoria da $12A4_{16}$ a $12A6_{16}$ sono riservate per mantenere i dati binari che devono essere interpretati come codici caratteri, allora il fatto che essi abbiano la stessa configurazione di cifre binarie delle parole binarie decimali codificate, non è importante. Finchè la logica del programma interpreta correttamente i contenuti delle parole di memoria, non possono verificarsi errori; e se la logica del programma non interpreta correttamente i contenuti delle parole di memoria, vuol dire che è errata e deve essere corretta.

SET DI CARATTERI

Per trattare testi è necessario un set completo di caratteri che comprende:

26 lettere minuscole
26 lettere maiuscole
circa 25 caratteri speciali (es. [+ / @ ! # , ecc.).
10 cifre numeriche

Questo set di caratteri arriva a 87 caratteri. Un gruppo di sei cifre binarie permette 64 combinazioni di cifre binarie 0 e 1 (2^6), che è insufficiente per rappresentare 87 caratteri. Un gruppo di sette cifre binarie permette 128 possibili combinazioni di cifre binarie 0 e 1, e quindi è sufficiente per le nostre necessità.

Il byte di 8 bit è stato universalmente accettato come l'unità dati per rappresentare i codici caratteri. I due codici caratteri più comuni elencati nell'Appendice A, sono conosciuti come l'American Standard Code for Information Interchange (ASCII) e l'Extended Binary Coded Decimal Interchange Coded (EBCDIC). Tutti i costruttori di mini e microcomputer usano l'ASCII.

PARITA'

Benchè, per rappresentare i caratteri, siano spesso sufficienti sette cifre binarie, ne vengono di solito usate otto. In questi casi, l'ottava

cifra binarie viene spesso usata per controllare gli errori, e si fa riferimento ad essa come al bit di parità; essa è posta a 1 o a 0, in modo che il numero di bit a 1 nel byte sia sempre dispari o sempre pari.

Se si usa il metodo dispari, verrà settato o risettato il bit di parità, in modo che il numero totale di bit a 1 sia sempre dispari. Ecco alcuni esempi:

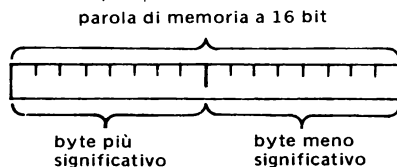
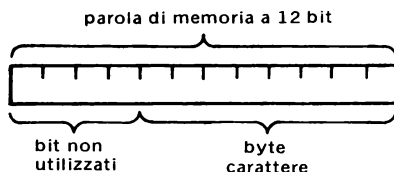
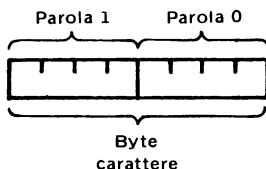
Se si usa il metodo pari, viene settato o risettato il bit di parità, in modo che il numero totale di bit a 1 sia sempre pari. Ecco alcuni esempi:

Il bit di parità si usa per avere la sicurezza che, da quando viene creato un byte carattere a quando esso viene riletto, non venga erroneamente cambiato nessun bit. Se, ad esempio, la parità è dispari, quando venga rilevato un numero pari di bit a 1 in un byte carattere, chiaramente il byte è errato. Analogamente, se si usa la parità pari, quando venga rilevato un numero dispari di bit a 1 in un byte carattere, il byte è certamente sbagliato.

Ecco un esempio di come dovrebbe essere memorizzato un messaggio in una sequenza di parole di memoria contigue usando codici caratteri ASCII con parità pari:

A questo punto sarebbe utile fare alcune osservazioni circa la parità e i codici per la correzione di errori.
Chiaramente, **il bit di ordine superiore di un byte può essere usato come bit di parità solo quando i contenuti del byte vengono interpretati come codici caratteri.** Se i contenuti del byte vengono interpretati come una qualunque forma di dati binari (codificati o no) il bit di ordine superiore è già specificato come una parte integrale del contenuto dei dati del byte; perciò questo bit non può essere usato come il bit di parità, quindi i dati binari, codificati e no, non possono avere il controllo di parità/disparità.

Si usano molti schemi elaborati, non solo per controllare che le cifre binarie non contengano errori, ma anche per rilevare quali siano questi errori e fare le correzioni del caso. Questi codici per la correzione degli errori non hanno niente a che fare, in particolare con i concetti riguardanti i mini e microcomputer e perciò, non ne parleremo in questo libro.

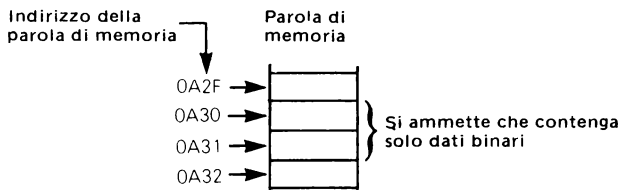
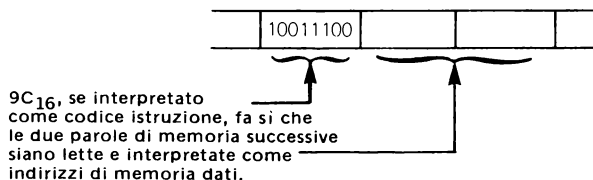
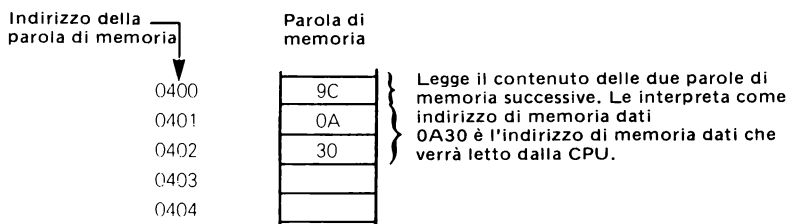
I microcomputer che hanno diverse lunghezze di parola da otto bit usano anch'essi un byte per rappresentare i codici caratteri. Un microcomputer di 16 bit metterà due byte in ogni parola di memoria, in questo modo:

Un microcomputer che usa parole di 12 bit memorizzerà il codice caratteri negli otto bit di ordine inferiore dei 12 e sprecherà i quattro bit di ordine superiore in questo modo:

Un microcomputer a 4 bit crea un byte usando due parole di memoria contigue:

I microcomputer a 4 bit, attualmente, sono quelli che si adattano meglio alle applicazioni BCD, come per esempio, quella dei calcolatori tascabili. Queste applicazioni considerano le unità di dati di 4 bit come entità univoche — una cifra BCD per ogni unità dati di 4 bit, il byte di 8 bit non è significativo, perciò il fatto che il microcomputer tratti sempre i dati in unità di 4 bit può semplificare notevolmente la programmazione.

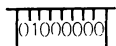
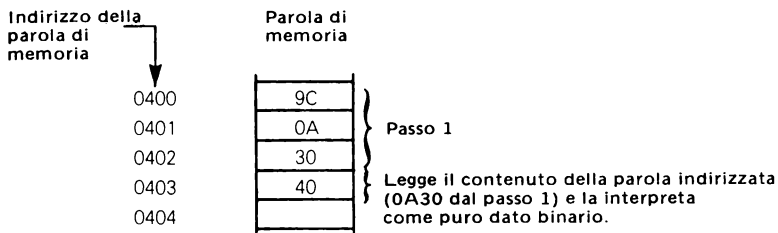
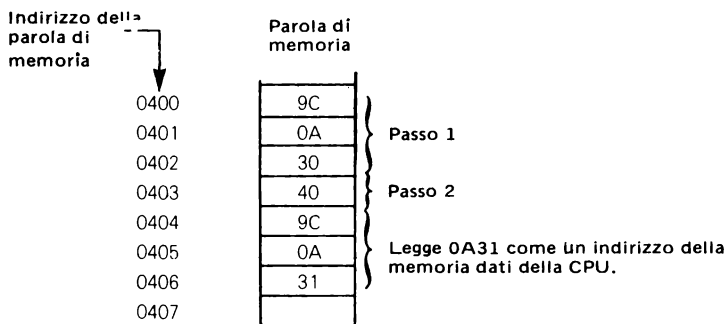
CODICI ISTRUZIONI

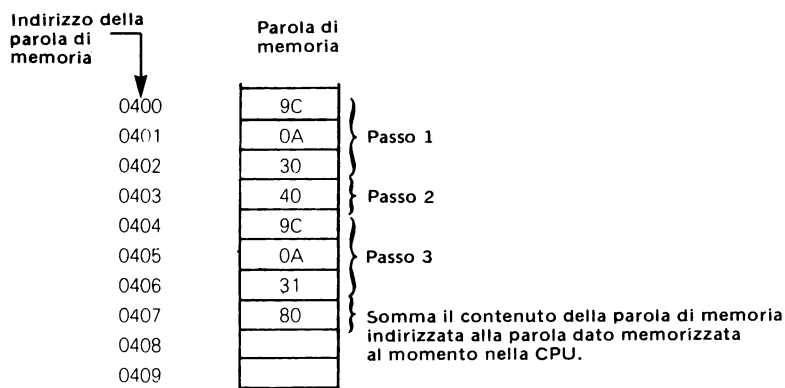
Le parole di memoria sono state finora interpretate come dati di una forma o di un'altra. I contenuti di una parola di memoria possono essere interpretati anche come un codice che identifichi una operazione richiesta al microcomputer.

Vediamo il semplice esempio di addizione binaria. Supponiamo che, il contenuto della parola di memoria con indirizzo 0A30 vadano aggiunto al contenuto della parola con indirizzo 0A31, e la somma vada memorizzata nella parola di memoria con indirizzo 0A31. I passi di programma per eseguire questa addizione binaria sono i seguenti:

1) **Identificare l'indirizzo della prima parola di memoria che va sommata.**
2) **Trasferire il contenuto di questa parola di memoria nel microcomputer.**
3) **Identificare l'indirizzo della seconda parola di memoria da sommare.**
4) **Aggiungere il contenuto di questa parola di memoria alla parola di memoria che è stata trasferita nel microcomputer al 2° passo.**
5) **Identificare l'indirizzo della parola di memoria in cui bisogna memorizzare il risultato.**
6) **Trasferire la somma in questa parola di memoria.**

La logica del programma specifica che le parole di memoria con indirizzi 0A30 e 0A31 devono contenere dati binari puri:

Supponiamo che i sei passi del programma debbano essere memorizzati in parole di memoria con indirizzi che iniziano a 0400: creeremo dei codici istruzioni per implementare l'addizione binaria in sei passi.
Il codice istruzione che identifica gli indirizzi di memoria occuperà tre byte, in questo modo:

Possiamo ora cominciare a costruire il programma nel seguente modo:

Il 2° passo richiede che il contenuto della parola di memoria indirizzata sia letto e interpretato come dato. Notate che non c'è bisogno che l'istruzione specifichi che tipo di dati contenga la parola di memoria, siete voi programmatore che dovete ricordare il tipo di dati contenuto nella parola indirizzata, e non cercate di non fare operazioni incompatibili con il tipo di dati. Finchè restiamo nell'ambito dei microcomputer, i dati sono dati binari puri — niente di più, niente di meno.

Supponiamo che il codice binario:

se interpretato come un'istruzione, faccia sì che il contenuto della parola di memoria indirizzata venga letto e interpretato come dato. Ora il nostro programma diventa:

Il passo 3 è una ripetizione del passo 1, bisogna solo specificare un indirizzo di memoria diverso ($0A31_{16}$). Ora il nostro programma diventa:

Il passo 4 è una variazione del passo 2; comunque anzichè leggere semplicemente il contenuto della parola di memoria dei dati indirizzati, la parola di memoria viene aggiunta, usando l'addizione binaria, alla parola di memoria precedentemente letta; supponiamo che questa operazione sia identificata dal codice istruzione:

Il nostro programma aumenta ora di un passo in questo modo:

Il passo 5 è una ripetizione del passo 3; l'indirizzo della parola di memoria in cui deve essere memorizzata la somma, $0A31_{16}$, è la parola di memoria indirizzata per ultima (nel passo 3), così non occorre ripetere l'istruzione per il passo 5. Procederemo invece al passo 6, e supporremo che la parola binaria:

se interpretata come codice istruzione, faccia sì che i dati vengano messi nella parola di memoria indirizzata per ultima. Il programma completo appare ora in questo modo:

Una volta che si può comunicare in qualche modo al microcomputer che esso troverà una sequenza di codici istruzione a partire dalla parola di memoria 0400_{16}, il fatto che ognuna di queste parole di memoria possa avere una configurazione di cifre binarie 0 e 1, che siano anche dati binari validi, o caratteri ASCII, è irrilevante.

Notate che il programma crea indirizzi di memoria che identificano parole di memoria che si suppone contengano dati binari puri. Si suppone che voi, come programmatore, vi assicuriate che queste parole di memoria contengono realmente dati binari. Se, per errore, vi sono istruzioni in qualche parte del programma che memorizzano codici caratteri in queste stesse parole di memoria, e poi vengono dati al computer i precedenti comandi, esso addizionerà due codici caratteri come se fossero dati binari. Saranno solo i risultati diversi da quelli previsti, a farvi capire che è stato commesso un errore.

Il fatto di illustrare un programma per microcomputer come un'addizione binaria in sei passi, è naturalmente solo per introdurre il discorso.
Come fa il microcomputer ad eseguire le operazioni richieste dal codice istruzione? A questa domanda risponderemo nel Capitolo 4.

Qual'è la richiesta di logica esterna da parte del microcomputer allo scopo di completare le operazioni specificate da un codice istruzione? A questa domanda risponderemo nel Capitolo 5.

Come si scrive un programma di un microcomputer? Questa domanda troverà risposta nel Capitolo 6.

Capitolo 4

L'UNITA' CENTRALE DEL MICROCOMPUTER

La logica di un microcomputer è implementata su uno o più chip. I chip sono impaccati in DIP, come mostrava la Fig. 1-1. La maggior parte delle DIP dei microcomputer hanno 40 pin, ma si possono trovare chip con diverso numero di pin, da 28 a 64.

CPU E MICROPROCESSORE

La cosa di cui potete essere sicuri è che uno dei valori DIP conterrà della logica a cui ci si riferisce come da un'Unità Centrale o "CPU" (Central Processing Unit); questo DIP è comunemente chiamato "microprocessore".
Il microcomputer può avere qualsiasi altra cosa, ma deve comunque avere una CPU.

La logica che costituisce una CPU può essere notevolmente diversa da un microcomputer ad un altro; è comunque necessario sottolineare queste differenze. In questo capitolo, il nostro scopo è di capire perchè è necessario.

Ricorderete dal Capitolo 3 che i contenuti di una parola di memoria possono essere interpretati in uno dei seguenti modi:

1) Come dati binari puri
2) Come dati binari codificati
3) Come un codice carattere
4) Come un codice istruzione.

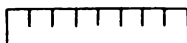
Questi quattro modi di interpretare il contenuto di una parola di memoria possono essere più generalmente suddivisi in due categorie: dati e istruzioni.
I dati binari puri, i dati binari codificati e i codici carattere hanno una cosa in comune: sono tutti dati. Sul contenuto delle parole di memoria si può operare sia isolatamente che in concomitanza col contenuto di altre parole di memoria.

I codici istruzione sono inseriti nella CPU come un mezzo per identificare la successiva operazione che volete fare eseguire dalla CPU. Una sequenza di codici istruzione, immagazzinati in memoria, costituiscono un programma.
Consideriamo il programma di addizione di sei passi descritto alla fine del Capitolo 3. Nei paragrafi successivi, esamineremo la logica che la CPU deve avere per eseguire questa addiziona binaria.

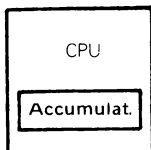
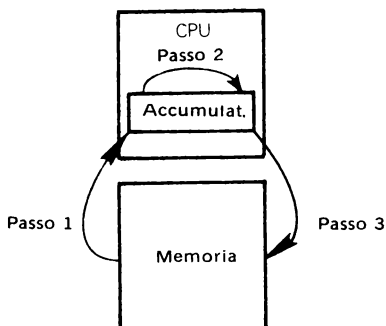
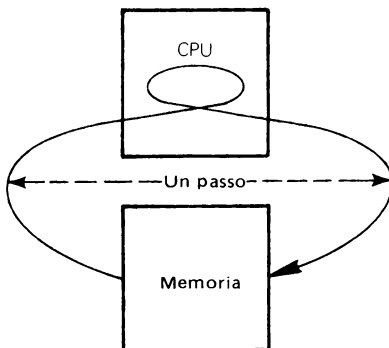
REGISTRI DELLA CPU

ACCUMULATORE

La CPU deve avere uno o più registri in cui i dati prelevati dalla memoria possono essere immagazzinati; chiameremo questi registri accumulatori. Dato che la maggioranza dei microcomputer usano una lunghezza di parola di 8 bit, questa è la lunghezza di parola che adotteremo noi — e supporremo di avere un accumulatore a 8 bit:

```
7 6 5 4 3 2 1 0   Numero di bit
┌─┬─┬─┬─┬─┬─┬─┬─┐
│ │ │ │ │ │ │ │ │  Accumulatore (A)
└─┴─┴─┴─┴─┴─┴─┴─┘
```

Per semplificare le cose, per il momento che vi sia un solo accumulatore nella CPU.

I dati presi al passo 2 del programma di addizione binaria descritto nel Capitolo 3 sono memorizzati nell'accumulatore.

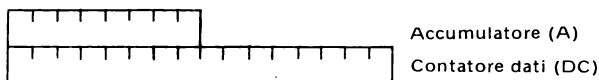
Generalmente la CPU opera sul contenuto di un accumulatore, e non accede direttamente alle parole di memoria. Ciò ha senso? Ricordate l'accumulatore è un registro all'interno della logica della CPU.

Siccome i dati vengono memorizzati in modo permanente nella memoria esterna, potete dedurre che, operando dati contenuti in un registro della CPU, si costringe il programma a definire una sequenza di istruzioni a tre passi:

mentre con un solo passo si sarebbe potuto fare così:

Sfortunatamente, non sempre è possibile utilizzare il passo qui illustrato. Alcune operazioni della CPU richiedono il contenuto di due parole di memoria per essere prese dalla memoria stessa:

Talvolta la CPU opera su di una sola parola:

Parola A ⟶ complemento a uno ⟶ Parola C

Si può anche considerare che l'operazione ad un solo passo è in qualche modo negativa: dobbiamo sempre perdere tempo prendendo dati dalla memoria, rimandando poi i risultati alla memoria stessa? La risposta è no. Ogni accesso alla memoria costa tempo e logica. Avendo nella CPU pochi registri di memorizzazione dati, possiamo avere un accesso alla memoria ogni cinque operazioni di CPU (più o meno); e questo è meglio che avere due accessi alla memoria per ogni operazione di CPU, cioè quanto richiede la sequenza ad un solo passo. Perciò quasi tutti i microcomputer hanno accumulatori o registri tipo accumulatori nella CPU.

CONTATORE DATI

Per accedere ad una parola di memoria, per leggere il contenuto o per memorizzarvi i dati, bisogna conoscerne l'indirizzo; questo indirizzo si trova in un registro che chiameremo il contatore dati (o Data Counter). La lunghezza del Data Counter dipenderà dalla quantità di memoria che il microcomputer può indirizzare. Qui illustriamo il 16 bit, che può indirizzare fino a 65.536 parole di memoria Data Counter.

La CPU di un microcomputer può avere uno o più Data Counter. Per semplificare le cose, supporremo, per il momento, che la CPU abbia un solo contatore dati.

Facendo nuovamente riferimento al programma di addizione binaria descritto nel Capitolo 3, gli indirizzi di memoria da $0A30_{16}$ a $0A31_{16}$ dovrebbero stare nel Data Counter.

Per accedere alla parola di memoria, la CPU ha bisogno di un accumulatore per memorizzare il contenuto della parola a cui ha fatto accesso, e di un Data Counter per memorizzare l'indirizzo della parola a cui accede.

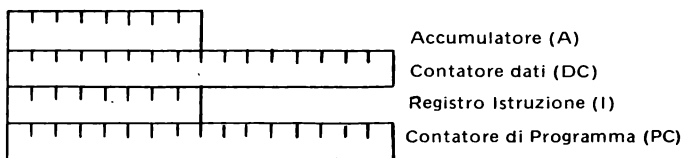
Analogamente, per trattare codici istruzione, la CPU avrà bisogno di un registro per memorizzare i codici istruzione, e di un registro per memorizzare l'indirizzo della parola da cui sta per essere preso il codice istruzione.

REGISTRO ISTRUZIONI

Il codice istruzione è memorizzato in un registro istruzioni; La CPU interpreterà sempre il contenuto del registro istruzioni come un codice istruzione.

PROGRAM COUNTER

L'indirizzo della parola di memoria da cui sarà preso il codice istruzione è fornito da un registro che chiameremo Program Counter o contatore di programma.

Il Program Counter è analogo al Data Counter con la sola differenza che il Data Counter indirizza sempre una parola di memoria contenente dati, mentre il Program Counter indirizza sempre una parola di memoria di programma. A questo punto abbiamo quattro registri:

Esiste una differenza concettuale importante fra il Data Counter e il Program Counter. **Se si memorizzano i codici istruzione, in parole di memoria successive, il problema di creare gli indirizzi del codice istruzioni nel Program Counter è risolto.** Tutto ciò che occorre è trovare un modo per caricare l'indirizzo iniziale nel Program Counter. Se, dopo l'accesso ad una parola di memoria per prendere un codice istruzione, il contenuto del Program Counter viene incrementato di 1, il Program Counter punterà alla parola di memoria che contiene il codice istruzione successivo.

Il Data Counter, invece, non ha di solito una serie di accessi in memoria sequenziali. Solo quando i dati vengono memorizzati in unità a più parole, o tabelle di dati, vengono poste in parole di memoria contigue, il Data Counter avrà necessità di accedere a posizione di memoria sequenziali. Anche in questo caso, non è chiaro se il Data Counter dovesse partire da un indirizzo di memoria basso e poi incrementarsi; o partire da un indirizzo di memoria alto e poi decrementarsi. Perciò, **la logica della CPU dovrà fornire all'utente del microcomputer una notevole flessibilità, quando esso deve modificare gli indirizzi nel Data Counter.**

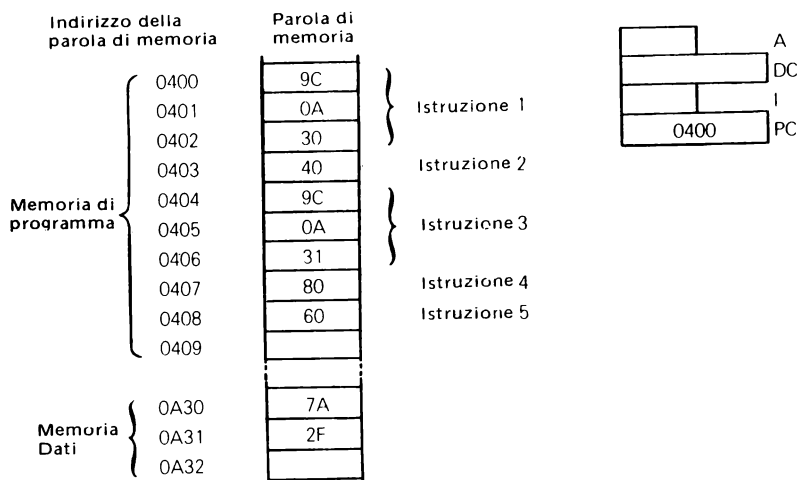
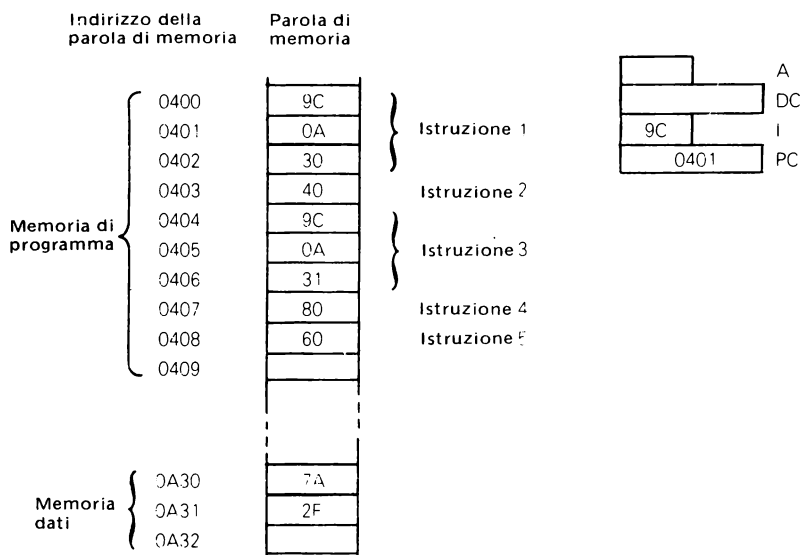
COME SI USANO I REGISTRI DELLA CPU

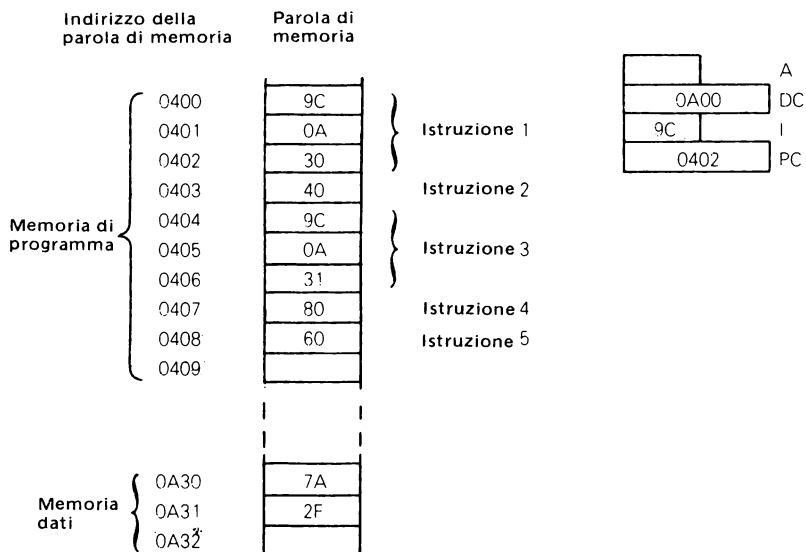
Per capire fino in fondo come si usano i registri nella CPU del microcomputer, rivedremo il programma dell'addizione binaria del Capitolo 3, facendo vedere come cambiano i contenuti dei registri.

ISTRUZIONI

D'ora in poi, ci riferiremo ad ogni passo del programma come ad un'istruzione, dato che in realtà ogni passo, come illustrato, identifica semplicemente il codice binario di un'istruzione.

Inizialmente il Program Counter (PC) contiene 0400_{16}, l'indirizzo di parola della prima istruzione nella memoria del programma; i contenuti di altri registri non sono no-

ti. Per completare l'illustrazione supponiamo, che le parole di memoria $0A30_{16}$ e $0A31_{16}$ contengono inizialmente $7A_{16}$ e $2F_{16}$, rispettivamente.

La CPU carica il contenuto della parola di memoria indirizzata dal PC nel registro istruzioni (I), assicurando così che il contenuto della parola di memoria sarà interpretato come un codice istruzione. La CPU incrementa poi il contenuto del PC:

Il codice 9C, che appare nel registro istruzioni, fa sì che la logica della CPU implementi due passi. Dapprima, il contenuto della parola di memoria indirizzata dal PC, viene

preso dalla memoria, ma viene memorizzato nel byte di ordine superiore del Data Counter (DC), la CPU incrementa poi il contenuto del PC:

Poi, il contenuto della parola di memoria indirizzata dal PC viene preso dalla memoria e memorizzato nel byte di ordine inferiore del DC. Di nuovo la CPU incrementa il contenuto del PC:

DATI DI TIPO IMMEDIATO

L'esecuzione dell'istruzione 1 è ora completa. Osservate che il contenuto della parola di memoria 0401_{16} e 0402_{16} è stato caricato nel registro DC, sebbene queste due parole di memoria siano nella memoria di programma e siano indirizzate dal Program Counter (PC). Il concetto importante è che il codice istruzione richiede che lo seguano immediatamente dei dati. I codici non-istruzione, richiesti dai codici istruzione e che appaiono immediatamente dopo di essi nella memoria di programma, si chiamano dati di tipo immediati.

Per esempio, nell'istruzione 1 le parole di memoria 0401_{16} e 0402_{16} contengono il dato immediato $0A30_{16}$. Il codice istruzione 9C, preso dalla parole di memoria 0400_{16}, identifica il modo in cui il dato immediato $0A30_{16}$ deve essere interpretato dalla CPU.

Passiamo ora all'istruzione 2. Avendo completato l'istruzione 1, la CPU prende il contenuto della parola di memoria indirizzata dal PC, poi incrementa il PC. Non essendo state date altre istruzioni specifiche il contenuto della parola prelevata è memorizzato nel registro 1, per essere interpretato come un codice istruzione:

Questo codice istruzione fa sì che la CPU prelevi il contenuto della parola di memoria indirizzata dal DC e carichi questa parola di memoria, nell'accumulatore (A), (vedi Figura 4-A).

Notate che nè il contenuto del DC nè quello del PC vengono incrementati. Il contenuto del PC non viene incrementato perchè 7A non è un dato immediato, ed è stato prelevato dalla memoria dati. Il contenuto del DC non viene incrementato dal momento che non si sa se si farà riferimento in maniera sequenziale alle parole dati.

E' stata completata l'esecuzione dell'istruzione 2 e il PC indirizza la parola di memoria di programma successiva, che contiene il codice istruzione dell'istruzione 3.

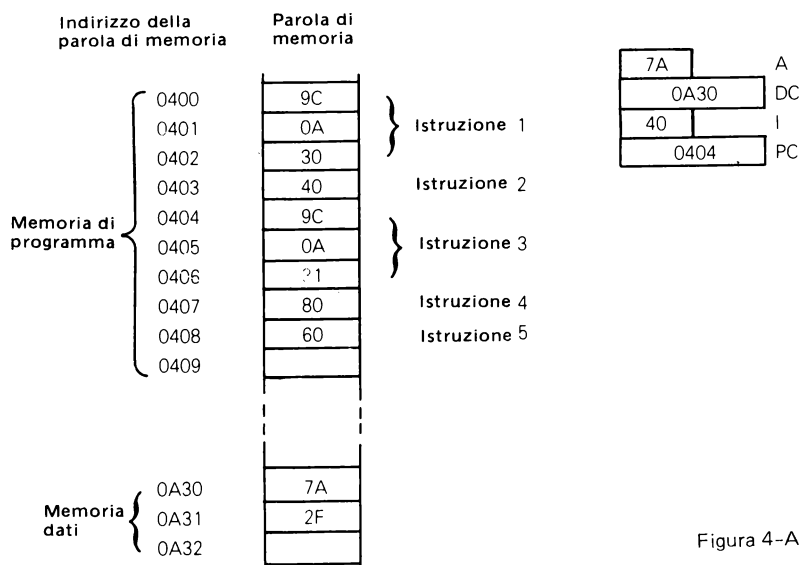



Figura 4-A.

L'istruzione 3 è una ripetizione dell'istruzione 1, tranne il fatto che il dato immediato $0A30_{16}$ è stato sostituto da $0A31_{16}$. Come per l'istruzione 1, i registri della CPU apportano cambiamenti in tre passi quando l'istruzione 3 è in corso, il passo 1 prende il codice istruzioni del registro I:

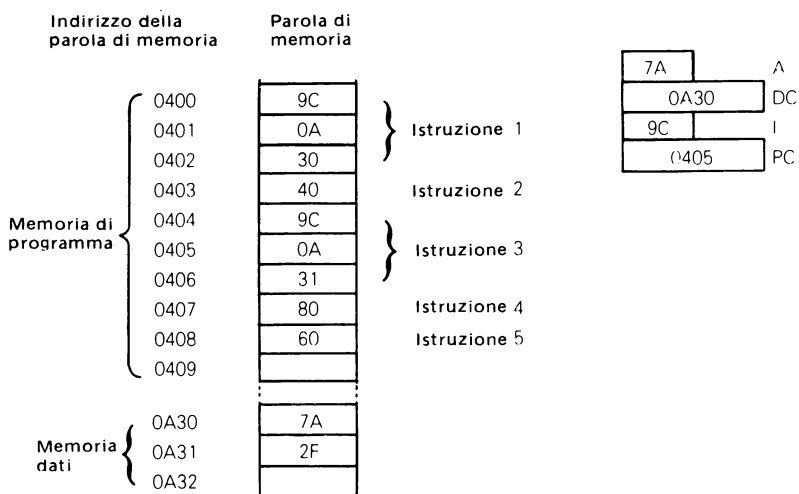
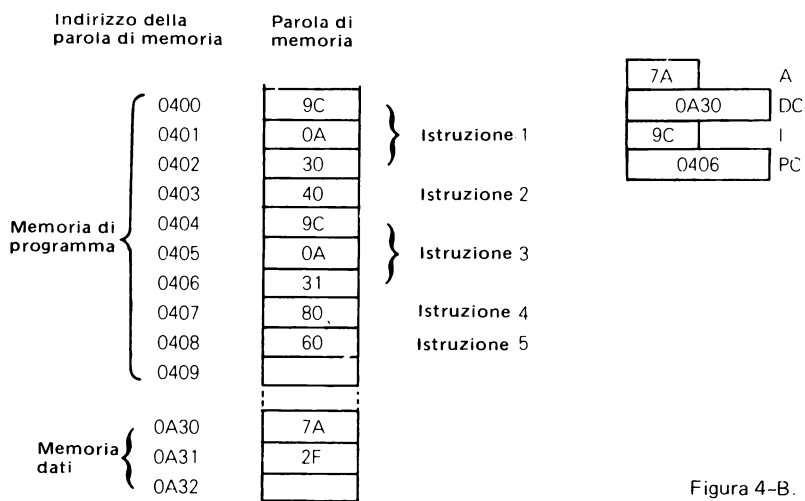
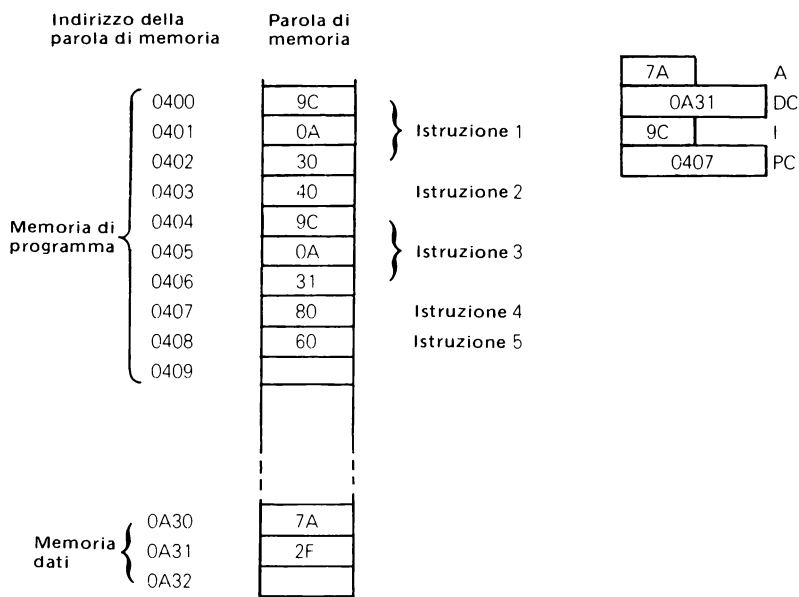
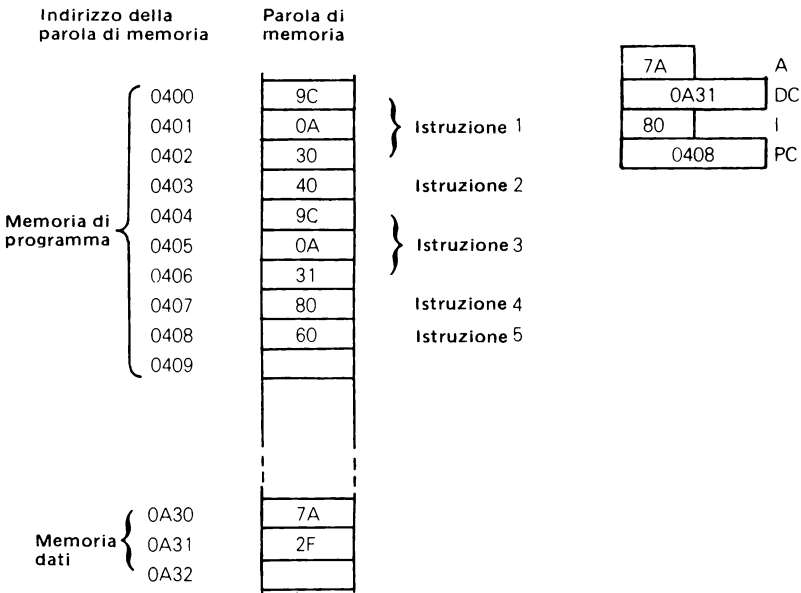
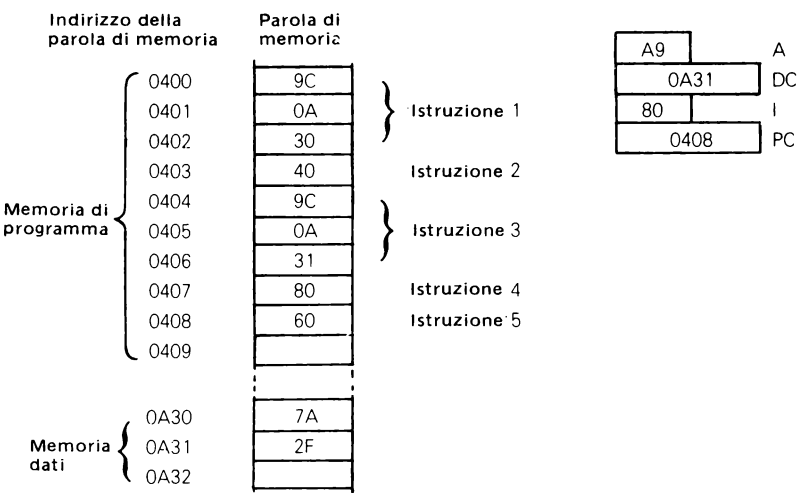



Figura 4-B.

Il passo 2 prende 0A dalla parola 0405_{16} e lo memorizza nel byte di ordine superiore del registro DC, che è per caso, quello che il registro DC già conteneva, così non sembra ci siano cambiamenti nel registro DC, (vedi Figura 4-B).

Il passo 3 cambia il byte di ordine inferiore del DC:

L'esecuzione dell'istruzione 3 ora è completata, e si può cominciare l'esecuzione della istruzione 4. Come con le istruzioni precedenti, la CPU inizia automaticamente caricando il contenuto della parola di memoria indirizzata dal PC in I:

Il codice istruzione 80 richiede alla CPU di prelevare il contenuto della parola di memoria indirizzata dal DC e di aggiungere questa parola dati al contenuto dell'accumulatore (A):

L'istruzione 4 è ora completata.

Se la somma in A fosse memorizzata in una parola di memoria diversa da $0A31_{16}$, dovremmo ora eseguire un'altra variazione dell'istruzione I per caricare un indirizzo di memoria nel DC. Ma il contenuto dell'accumulatore deve essere memorizzato nella parola di memoria $0A31_{16}$, e questa è la parola di memoria attualmente indirizzata dal DC, perciò non è necessaria un'istruzione di caricamento di un indirizzo di memoria. Passiamo direttamente all'istruzione 5, che memorizza il contenuto dell'accumulatore nella parola di memoria $0A31_{16}$ per mezzo di questi 2 passi:

Passo 1 : prelevare il codice istruzione come al solito;

Passo 2 : memorizzare il contenuto dell'accumulatore nella parola di memoria indirizzata dal DC:

L'istruzione 5 ha completato l'esecuzione, e il programma è fatto.

L'UNITA' ARITMETICO-LOGICA

Le reali manipolazioni dei dati all'interno della CPU sono gestite da un gruppo di componenti logici discreti a cui ci si riferisce collettivamente come all'Unità Aritmetico-Logica (ALU). Un ALU deve essere capace di operare su dati binari in incrementi di parole di memoria; in altre parole, l'ALU di un microcomputer a 8 bit opererà su unità di dati 8 cifre binarie. La ALU deve avere la logica necessaria per eseguire le seguenti operazioni:

1) Addizione binaria

2) Operazioni booleane

3) Complementare una parola dati

4) Far scorrere (shift) una parola dati di un bit verso destra o verso sinistra.

Con questi pochi elementi di logica nella ALU, si possono eseguire tutte le operazioni di gestione dati più complesse, che può richiedere la CPU.

L'UNITA' DI CONTROLLO

E' l'unità di controllo (CU) che mette in sequenza gli elementi logici della ALU allo scopo di implementare qualunque operazione richiesta. L'unità di controllo è guidata dal contenuto del registro istruzioni. In altre parole, il contenuto del registro istruzioni e decodificato dall'unità di controllo. In risposta alla configurazione dei bit del codice istruzione, l'unità di controllo genera una sequenza di segnali di abilitazione per far fluire i dati in modo appropriato attraverso la ALU e per abilitare i moduli logici della ALU al momento opportuno. **La CPU del microcomputer che ne risulta è illustrata in Figura 4-1.**

Il "registro buffer" contiene i dati che si transitano per la CPU. Per esempio, quando si sommano due byte dati (come nell'istruzione 4 dell'esempio di addizione binaria), la parola che è prelevata dalla memoria, per essere sommata al contenuto dell'accumulatore, sarà memorizzata nel registro buffer.

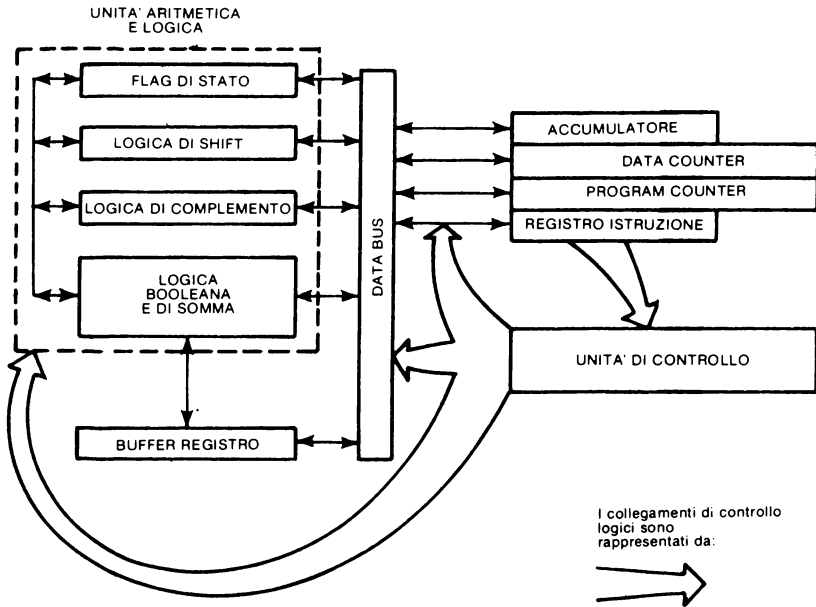
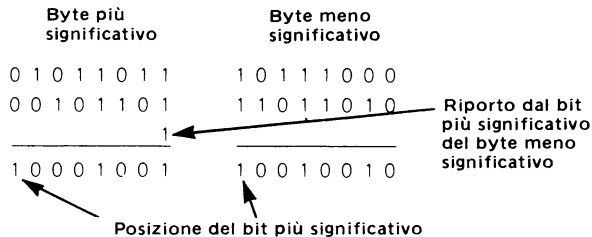



Figura 4-1. Rappresentazione funzionale di un'Unità di Controllo

FLAG DI STATO

Una CPU deve avere un set di gate logici ad una sola cifra binaria che vengono settati o risettati automaticamente per riflettere i risultati dell'operazione svolta nella ALU. Ognuno di questi gate logici a cifre binarie si chiama flag di stato.

STATO CARRY

Abbiamo già incontrato due flag di stato nel Capitolo 3: il Carry e il Carry Intermedio. Per eseguire operazioni aritmetiche a più byte, qualunque Carry del bit di ordine superiore di due parole dati deve essere registrato nello stato Carry, in modo che possa essere diffuso nelle parole di memoria di ordine superiore:

Lo stato Carry è utile anche quando si eseguono operazioni di shift di più parole, come descritto nel Capitolo 6.

STATO CARRY INTERMEDIO

Per eseguire operazioni aritmetiche BCD, è necessario anche registrare i carry dei quattro bit di ordine inferiore di un'unità di 8 bit, dato che, come descritto nel capitolo 2, ogni unità di 4 bit di un byte codifica una diversa cifra decimale.

Vi sono degli stati aggiuntivi che possono rivelarsi utili anche quando si eseguono vari tipi di manipolazioni di dati o operazioni decisionali.

Un flag di stato, detto zero, può essere settato a 1, per indicare che un'operazione ha generato un risultato pari a zero; in ogni altro caso questo flag sarà risettato a 0.

STATO ZERO

Su questo punto occorre prestare un attimo di attenzione. La maggior parte dei micro e minicomputer hanno un flag di stato Zero. E' universalmente accettato che il flag di stato Zero sia settato a 1 se un'operazione genera un risultato zero, mentre il flag di stato è settato a 0 per un risultato non-zero. In altre parole, il flag di stato è universalmente posizionato al complemento della condizione del risültato.

QUANDO SI MODIFICANO GLI STATI

C'è un altro punto da rilevare a proposito del flag di stato Zero, e della maggior parte degli altri flag di stato. **I progettisti di microcomputer selezionano accuratamente le istruzioni che settano o risettano flag di stato e quelle che non lo fanno.**

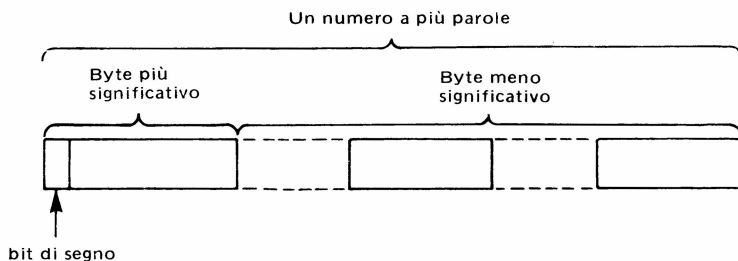
Consideriamo il caso più che ovvio dell'addizione a più byte, come precedentemente illustrato. Si sommano le parole di ordine inferiore dei due numeri a 2 parole, e lo stato Carry viene settato o risettato per riflettere un eventuale riporto del bit di ordine superiore delle parole di ordine inferiore. Il Carry deve essere aggiunto al bit di ordine inferiore delle due parole di ordine superiore dei numeri a due parole. Ciò significa che lo stato Carry deve essere conservato, mentre le due parole di ordine superiore vengono caricate nei registri della CPU. Chiaramente sarebbe disastroso progettare la logica in modo che il caricamento di una parola dalla memoria, azzeri lo stato Carry per riflettere il fatto che l'operazione di carico non ha generato un riporto.

A questo punto, è importante solo ricordare che non tutte le istruzioni influenzeranno tutti i flag di stato; inoltre, il modo con cui i flag di stato vengono settati o risettati è molto importante ed è una delle più importanti caratteristiche di tutti i progetti di CPU di un microcomputer. In altre parole, i flag di stato non rappresentano necessariamente le condizioni del momento all'interno della CPU; essi possono rappresentare il risultato dell'ultima operazione chiave eseguita.

STATO DI SEGNO

L'uso del bit di ordine superiore di una parola di memoria come bit del segno, quando si eseguono operazioni binarie con segno, dà luogo a due flag di stato. Primo, lo stato Segno, che è semplicemente il contenuto del bit del segno (o il suo complemento). Il flag di stato Segno fa si che si possano eseguire test per numeri positivi o negativi quando le parole di memoria devono essere interpretate come dati binari con segno.

Il bit del segno è sempre il bit di ordine superiore di qualsiasi numero formato da una singola parola o da più parole.

Nel microcomputer, comunque, il bit di ordine superiore di ogni byte sarà considerato come bit del segno. E' la logica di programma che deve decidere quando ignorare lo stato Segno e quando interpretarlo.

STATO DI OVERFLOW

Vi è poi lo stato Overflow. Ricordate che la CPU del microcomputer tratterà tutte le addizioni binarie nello stesso modo — come addizioni binarie pure. Se viene generato un Carry sommando due parole di ordine inferiore di due numeri a più parole, allora Carry è vero e riflette semplicemente un riporto alla parola superiore successiva della somma. Potete vedere quanto detto nell'addizione di due numeri di 4 parole, con 8 bit per parola:

La CPU del microcomputer non ha modo di sapere se una parola di memoria è un'entità numerica singola o è parte di un'entità numerica a più parole e, nel caso sia parte di un'entità numerica a più parole, se è nel mezzo della parola o ad una estremità. In questo caso, un Carry generato come il risultato di un'addizione, per la logica del microcomputer, è sempre valido.

Quando il bit di ordine superiore di una parola viene interpretato come un bit del segno, qualunque Carry del penultimo bit rappresenterà un errore di overflow, cioè un risultato che non può essere contenuto nello spazio predisposto. Consideriamo una sola parola di 8 bit che viene interpretata come dato binario con segno:

Dobbiamo trovare un modo per identificare i risultati sbagliati dell'addizione binaria con segno.
Un riporto del bit di ordine superiore del dato segnala sempre un errore? A dire il vero no; vediamo: (-2) + (-2) = (-4).

Sebbene vi sia un Carry dal bit di ordine superiore del dato, il risultato è -4, che è esatto. Useremo il simbolo C_S per rappresentare un Carry dal bit di segno e C_P per rappresentare un Carry dal bit di ordine superiore del dato. Che cosa succede se C_S e C_P sono entrambi 0?

```
   2       0 0 0 0 0 0 1 0
  +2       0 0 0 0 0 0 1 0
  =4       0 0 0 0 0 1 0 0

      Cp=0 ← Cs=0
```

```
        11110100      (-0C)
      + 00001001      +  09
      = 11111101      =(-03)

 Cs = 0 ←    Cp = 0
```

Finchè sia C_S che C_P sono zero, la risposta è sempre esatta.

Ora vediamo alcuni esempi in cui C_S e C_P sono entrambi 1:

```
   1 0 0 0 1 0 1 1      (-75)
  +0 1 1 1 1 0 0 1      +  79
   0 0 0 0 0 1 0 0      =(+ 4)

 1 ← 1
 Cs  Cp
```

(Ricordiamo che 10001011 è -75_{16} perchè $+75_{16}$ è 01110101, il complemento a due del quale è 10001011).

```
      1 1 0 1 1 0 0 0      ( -28)
     +0 1 0 1 1 0 0 1     +  59
      0 0 1 1 0 0 0 1     =(+31)

  1 ←  1
  Cs   Cp
```

```
      1 1 0 0 0 1 1 1      (-39)
     +1 1 1 0 0 1 1 0     +(-1A)
      1 0 1 0 1 1 0 1     =(-53)

  1 ←  1
  Cs   Cp
```

Quando C_S e C_P sono entrambi 1, la risposta è sempre esatta.

Quando C_S e C_P sono diversi, cioè uno o l'altro, ma mai entrambi, sono 1, la risposta è sempre sbagliata:

```
      0 1 0 0 0 1 0 1        45
      0 1 1 0 0 1 1 1     +  67
      1 0 1 0 1 1 0 0     =(-54) ?

  0 ←  1
  Cs   Cp
```

```
      1 0 0 1 0 0 1 0      (-6E)
      1 0 1 0 0 1 0 0     +(-5C)
      0 0 1 1 0 1 1 0     =  +36 ?

  1 ←  0
  Cs   Cp
```

STRATEGIA DI POSIZIONAMENTO DELLO STATO OVERFLOW

La nostra strategia per settare o risettare lo stato Overflow è quindi chiara. Quando i Carry del bit del segno e del penultimo bit sono uguali (C_P e C_S sono entrambi 0 o entrambi 1), lo stato Overflow verrà settato a zero. Quando questi due Carry sono diversi, lo stato Overflow sarà settato a 1, indicando che la risposta ha superato lo spazio disponibile, ed è quindi sbagliata.

Enunciato diversamente, **l'Overflow sarà l'OR esclusivo dei Carry del bit del segno e del penultimo bit:**

$$\text{OVERFLOW} = C_S \oplus C_P$$

STATO DI PARITA' **Lo stato parità è l'ultimo stato di cui vale la pena di parlare. Questo flag, se presente, è settato a 1 ogni volta che un'operazione di trasferimento dati rivela un byte di dati con la parità sbagliata.** Chiaramente questo stato verrà quasi sempre ignorato, dato che esso assume significato solo quando i contenuti di una parola di memoria vengono interpretati come un codice caratteri.

ESECUZIONE DELLE ISTRUZIONI

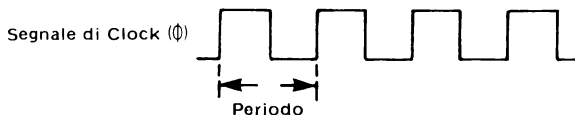
Abbiamo descritto come la CPU di un microcomputer può interpretare il contenuto di una parola di memoria come un codice istruzioni. Ma ciò lascia una serie di domande senza risposta. Qual'è il numero massimo o minimo di passi logici che costituiscono un'istruzione? Che cosa accade all'interno della CPU durante l'esecuzione di una istruzione? E quale supporto di logica esterna richiede la CPU?
Nel rispondere a queste domande, presentiamo un concetto molto importante sui microcomputer, ed una delle differenze chiave fra i mini e i microcomputer.

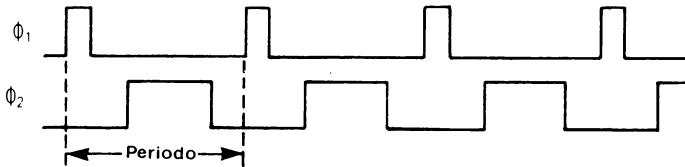
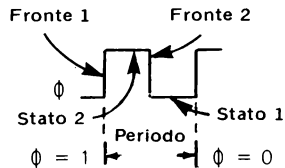
Nel mondo dei minicomputer, la caratteristica più importante da ricercare in un set di istruzioni è la versatilità di operazioni eseguite dalla CPU in risposta ad ogni codice istruzione. Tale versatilità è discreta nei minicomputer; essi vengono spesso usati per eseguire vari e svariati compiti, e la programmazione può essere la spesa maggiore. Nel mondo di microcomputer, questa domanda è molto più importante: qual'è la necessità di logica esterna da parte della CPU?

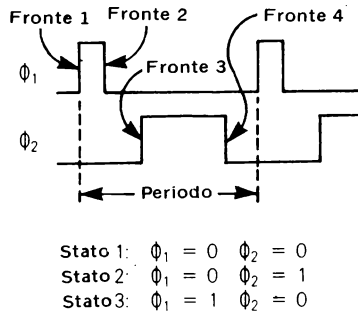
Le istruzioni più complicate richiedono di solito una logica complessa esterna alla CPU. Questo non riguarda l'utente del minicomputer che compra la CPU insieme a tutta la logica esterna, in un'unica confezione. Interessa parecchio invece l'utente che interfaccia la sua logica, spesso direttamente con la CPU. Anche se un Microcomputer ha un costo che varia 5$ a 100$, svanirà la convenienza economica dell'uso se la logica di interfaccia richiesta non è poco costosa.

Vedremo adesso come si esegue un'istruzione, e poi, alla fine di questo capitolo, ritorneremo sulle differenze fra mini e microcomputer.

TEMPORIZZAZIONE O TIMING DELLE ISTRUZIONI

Come tutta la logica digitale, le operazioni all'interno della CPU di un microcomputer sono controllate da un clock al cristallo, con un periodo che può variare da 100 nanosecondi ad un microsecondo. Faremo riferimento a questo segnale di temporizzazione usando il simbolo Ø:

Mentre il cristallo deve essere esterno al chip della CPU, la logica che genera il segnale di Timing può essere o non essere sullo stesso chip della CPU. Inoltre, a seconda di come la CPU è stata progettata, il segnale di Timing può essere un solo segnale diretto, come precedentemente illustrato, o può consistere di un'interazione più complessa di segnali.

Ecco una possibile combinazione di due segnali, identificati dai simboli Φ_1 e Φ_2:

Il segnale semplice fornisce due fronti e due stati per ogni periodo:

Il segnale più complesso fornisce quattro fronti e tre stati per ogni periodo:

Stato 1: $\Phi_1 = 0 \quad \Phi_2 = 0$
Stato 2: $\Phi_1 = 0 \quad \Phi_2 = 1$
Stato 3: $\Phi_1 = 1 \quad \Phi_2 = 0$

In questo capitolo useremo il segnale semplice Φ.

CICLI DI ISTRUZIONI

In qualunque microcomputer, l'esecuzione di ogni istruzione può essere divisa in due parti: il prelevamento e l'esecuzione dell'istruzione. Ne abbiamo già parlato in questo Capitolo a proposito dell'esempio di addizione binaria in sei passi. Ricordate che ogni istruzione inizia quando il codice istruzione viene caricato nel registro istruzioni. Ci riferiremo a questa operazione come al prelevamento dell'istruzione.

PRELEVAMENTO DELL'ISTRUZIONE

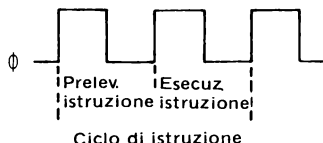
Durante il prelevamento dell'istruzione, la logica della CPU mette in output il contenuto del Program Counter, insieme con gli appropriati segnali di controllo che specificano che la logica esterna deve rimandare il contenuto della parola in memoria indirizzata dal Program Counter. Finchè si ha a che fare con la logica esterna, questa è semplicemente un'operazione di lettura.
Una volta ricevuto dalla CPU il contenuto della parola di memoria viene memorizzato nel registro istruzioni, e così è interpretato come un codice istruzione.

Mentre la logica esterna risponde al prelevamento dell'istruzione, la CPU usa la propria logica interna per raggiungere 1 al contenuto del Program Counter, che punta ora sulla parola di memoria seguente a quella da cui è stato preso il codice istruzione in esecuzione.

ESECUZIONE DELL'ISTRUZIONE

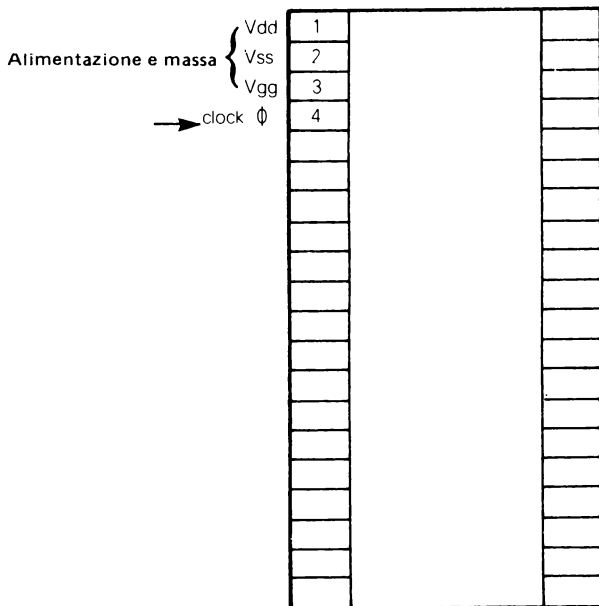
Una volta che il codice istruzione è nel registro istruzioni, fa scattare una sequenza di eventi controllati dall'unità di controllo; questa sequenza costituisce l'esecuzione dell'istruzione.

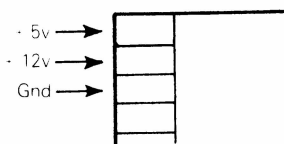
Per eseguire un'istruzione si usano due periodi di Timing; uno seguirà il tempo di prelevamento dell'istruzione, l'altro dell'esecuzione dell'istruzione stessa:

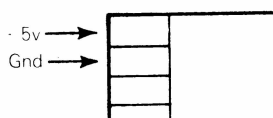
Ciclo di istruzione

PIN E SEGNALI DELLA CPU

Consideriamo ora i segnali per mezzo dei quali la CPU comunica con la logica esterna. Il DIP a 40 pin, essendo il più usato per i microcomputer attuali, è quello che adotteremo noi — questo significa che si possono mettere in input/output 40 segnali, compresi il clock, l'alimentazione e la massa.

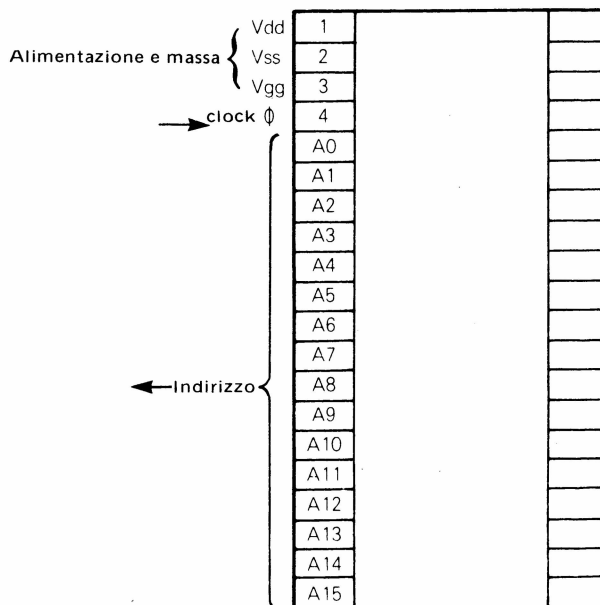
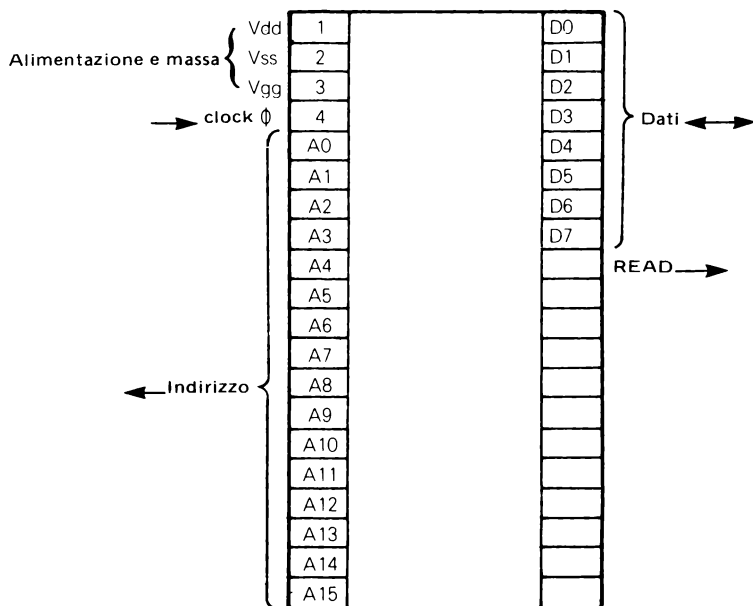
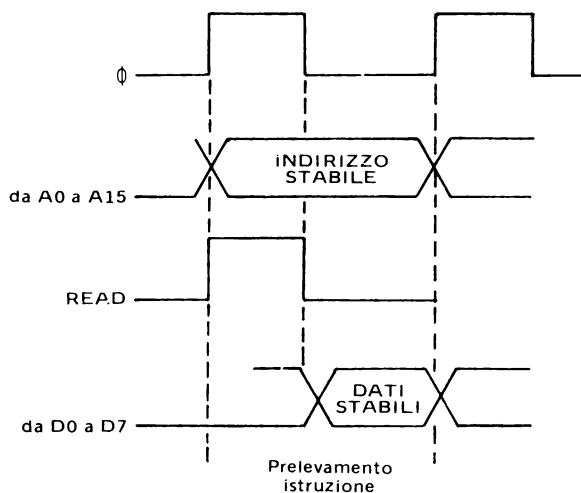
Il modo con cui vengono usati i 40 pin del DIP costituisce una delle caratteristiche che più variano da microcomputer a microcomputer, ma tutti hanno gli stessi pin iniziali.

Vdd è il canale di scarico della corrente, o ingresso di alimentazione.
Vss è la sorgente di corrente, o massa.
Vgg è il voltaggio dei gate, non è necessario in tutti i dispositivi LSI.

ALIMENTAZIONE

Molti dispositivi hanno un collegamento con l'alimentazione molto più semplice, cioè in uno dei seguenti modi:

MASSA (GND)

In questo caso il dispositivo ha due alimentazioni di corrente, +5V e +12V, più una sola massa. Spesso basta una sola alimentazione di corrente:

PRELEVAMENTO DELLE ISTRUZIONI: SEGNALI E TIMING

Nell'eseguire una qualsiasi istruzione, il primo passo è il prelevamento dell'istruzione stessa; cioè, in realtà, una lettura della memoria. Questo esige che un indirizzo di memoria venga messo in output, e che in risposta venga presa in input una parola dati.

Se l'indirizzo di memoria può variare da 0 a 65.535, saranno necessari sedici pin d'indirizzamento, uno per ogni cifra binaria dell'indirizzo:

Tutti i dati entreranno e lasceranno la CPU del microcomputer per mezzo di otto segnali bidirezionali. Un segnale di controllo READ, che indica che i dati devono essere presi in input nella CPU, completa le richieste per il prelevamento delle istruzioni.

Il seguente diagramma di Timing definisce la sequenza del prelevamento di una istruzione, così come viene controllata dalla CPU.

Rivolgeremo ora la nostra attenzione alle cose necessarie all'esecuzione di una istruzione Consideriamo il programma di addizione binaria in sei passi che è stato descritto alla fine del Capitolo 3. Vi sono quattro tipi di istruzione separati e distinti all'interno del programma. Essi sono:

1) **Caricare un indirizzo di memoria nel Data Counter (istruzione 1 e 3).**
2) **Prendere il contenuto della parola indirizzata dal Data Counter e memorizzarlo nell'accumulatore (istruzione 2).**
3) **Prendere il contenuto della parola indirizzata dal Data Counter, sommarlo al contenuto dell'accumulatore, e memorizzare il risultato nell'accumulatore (istruzione 4).**
4) **Memorizzare il contenuto dell'accumulatore nella parola di memoria indirizzata dal Data Counter (istruzione 5).**

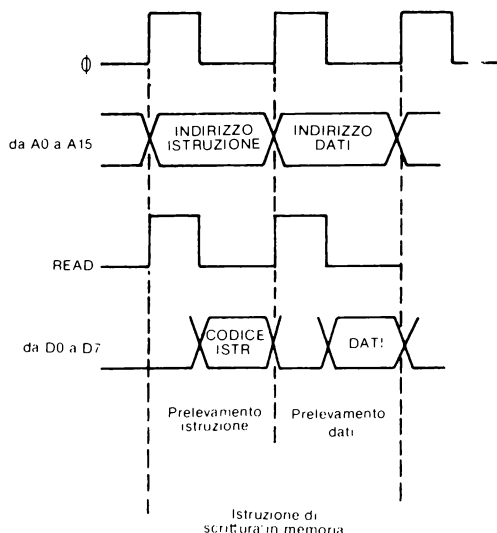
Esaminiamo ora in che modo si eseguono tutti questi tipi di istruzioni, in termini di cicli di istruzioni e segnali di controllo messi in output e presi in input nella CPU.

LETTURA DELLA MEMORIA: SEGNALI E TIMING

L'istruzione 2 è la più semplice, perciò inizieremo con lei. Come tutte le istruzioni comincia con il prelevamento dell'istruzione stessa.

L'unita di controllo decodifica il codice istruzione 40_{16}, e in risposta fa sì che una parola venga prelevata dalla memoria. In realtà, siccome i dati sono in una parola contenuta in un dispositivo di memoria, tutto quello che la CPU può fare è generare segnali ai suoi pin; è la logica esterna alla CPU che deve rispondere a questi segnali se c'è da eseguire un prelevamento di dati.

Come abbiamo visto per la logica esterna, i segnali generati dalla CPU per prelevare i dati sono identici ai segnali generati per prelevare un'istruzione.

Perciò **il Timing per un'istruzione di lettura in memoria è il seguente:**

Queste sono le uniche differenze tra il prelevamento di un'istruzione e i cicli di prelevamento di un dato:

1) Durante il ciclo di prelevamento di un'istruzione, l'indirizzo messo in output su A0 - A15 è il contenuto del registro PC. Durante il ciclo di prelevamento di un dato, esso è il contenuto del Data Counter.

2) Durante il prelevamento di un'istruzione, il dato in input è memorizzato nel registro; durante il prelevamento di un dato è memorizzato nell'accumulatore.

NECESSITA' DI LOGICA ESTERNA

Questo semplice schema richiede pochissima logica esterna. Se il segnale READ è alto quanto è alto Φ allora i circuiti della memoria devono codificare A0–A15. Il modulo di memoria selezionato deve estrarre il contenuto della parola di memoria indirizzata e assicurarsi che essa si trovi ai pin dati del microcomputer quando Φ è basso.

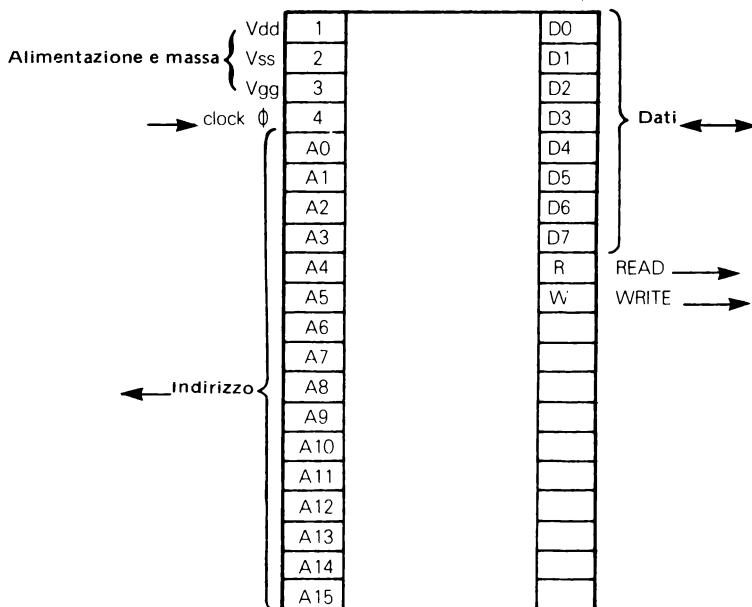
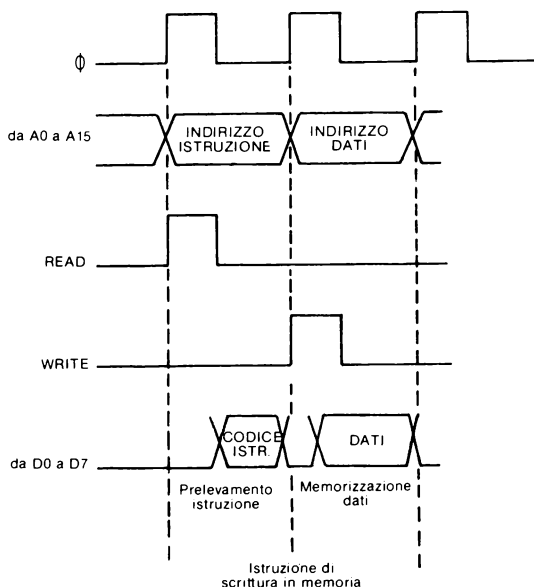
La richiesta di logica esterna da parte della CPU del microcomputer durante un'operazione di lettura è una richiesta di logica semplice, standard che fa parte di qualunque dispositivo di memoria; ma rimandiamo questo argomento al Capitolo 5, e continuiamo con i segnali e il Timing della CPU per l'istruzione ADD.

OPERAZIONE ADD: SEGNALI E TIMING

Per eseguire un'addizione, la CPU prende il contenuto di una parola di memoria, esattamente come per una istruzione di lettura di memoria; comunque, per l'istruzione ADD, i dati prelevati sono sommati al contenuto dell'accumulatore; mentre i dati prelevati durante un'istruzione di lettura in memoria sono depositati così come si trovano nell'accumulatore.

Come abbiamo visto, per la logica esterna, non vi è differenza fra i segnali generati per un'istruzione ADD o un'istruzione di lettura in memoria.

SCRITTURA IN MEMORIA: SEGNALI E TIMING

L'istruzione 4 fa sì che il contenuto dell'accumulatore sia memorizzato nella parola di memoria indirizzata dal Data Counter; questa si chiama istruzione di scrittura in memoria. **Come abbiamo visto, per la logica esterna, l'unica differenza fra le sequenze di segnali per una scrittura in memoria ed una lettura in memoria è che il segnale WRITE deve essere alto, invece del segnale READ, quando Φ è alto. Dobbiamo quindi aggiungere un segnale WRITE al dispositivo della nostra CPU.**

Il Timing per un'istruzione di scrittura in memoria è il seguente:

CARICAMENTO DATA COUNTER: SEGNALI E TIMING

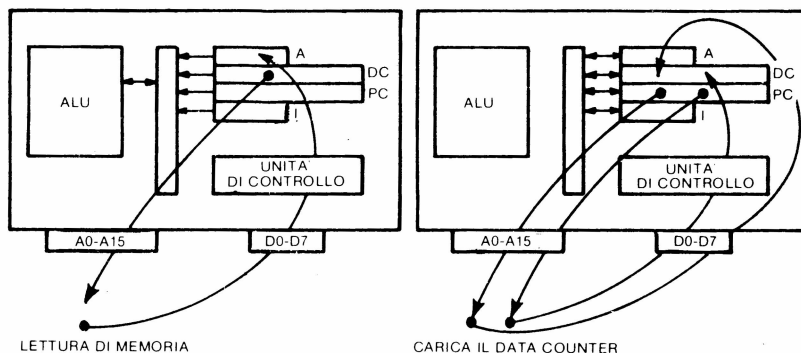
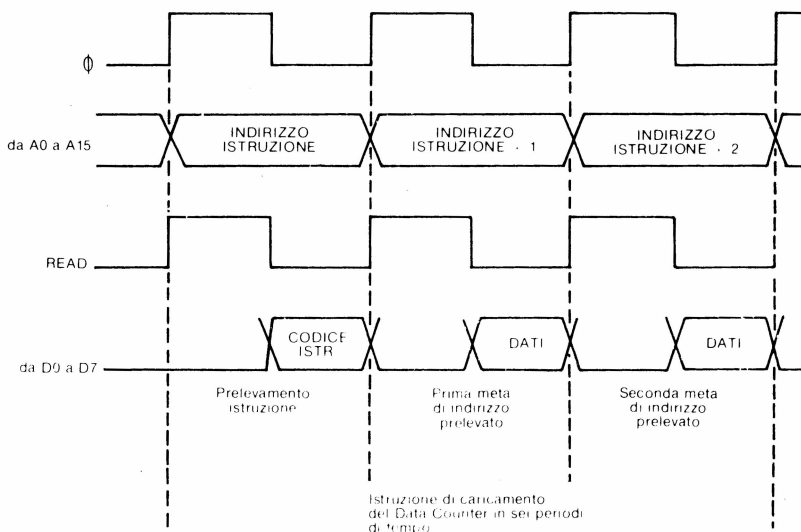
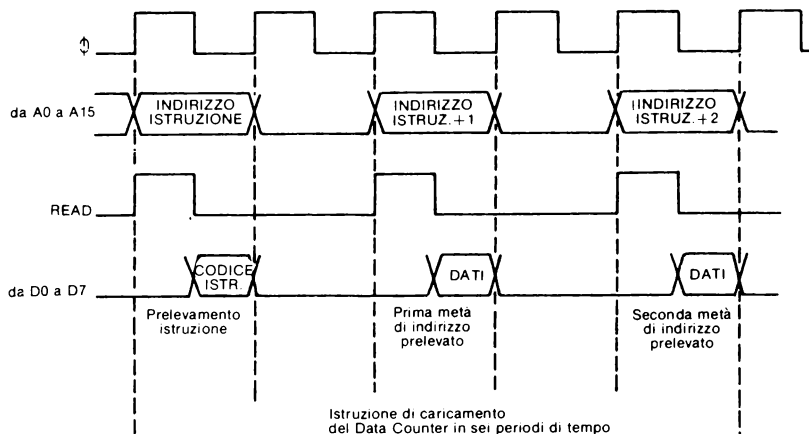
L'istruzione 1 carica un indirizzo di memoria nel Data Counter. Questa istruzione occupa tre parole di memoria una per il codice istruzione e altre due per l'indirizzo di memoria.

Notate che l'istruzione di caricamento del Data Counter è equivalente a due lettere in memoria, con queste differenze:

1) Entrambe le letture in memoria per il caricamento del Data Counter sono specificate da un codice istruzione $9C_{16}$. Al contrario, il codice dell'istruzione di lettura in memoria 40_{16} dà il via ad una sola lettura in memoria.

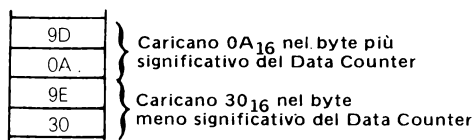
2) L'istruzione di caricamento del Data Counter preleva i dati dalle parole di memoria i cui indirizzi vengono dal Program Counter. Per un'istruzione di lettura in memoria, il Data Counter fornisce l'indirizzo di memoria dei dati.

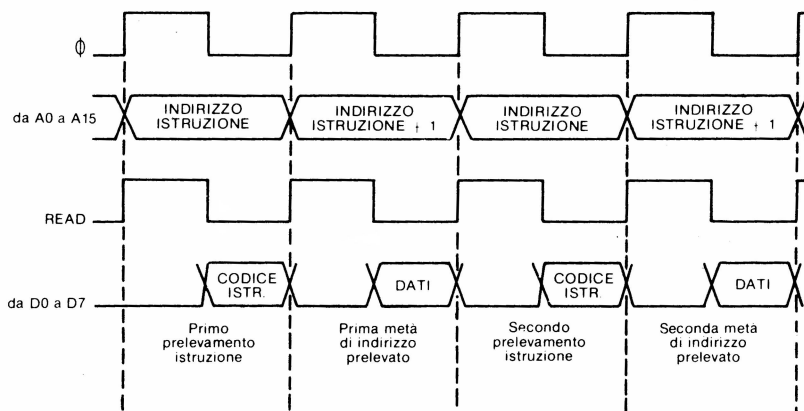
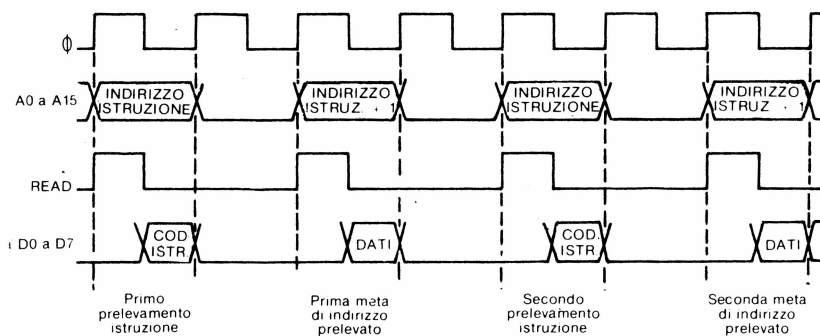
3) La lettura dei dati dalla memoria da parte dell'istruzione di caricamento del Data Counter è memorizzata nelle metà superiori e inferiori del Data Counter. Per una istruzione di lettura in memoria, i dati prelevati sono memorizzati nell'accumulatore.

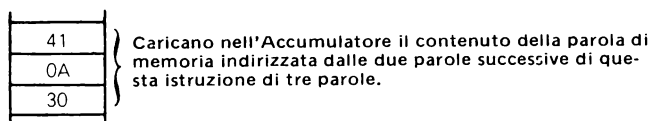
Le differenze fra le istruzioni di caricamento del Data Counter e di lettura in memoria possono essere evidenziate nel modo seguente:

Mentre le operazioni logiche interne alla CPU sono completamente diverse per le istruzioni di caricamento del Data Counter e di lettura in memoria, le sequenze esterne di segnali e di Timing sono notevolmente simili. La CPU di un microcomputer molto "intelligente" potrebbe eseguire l'istruzione di caricamento del Data Counter in tre periodi di tempo, come segue:

Alcuni computer non cercano di essere tanto intelligenti. Per semplificare la logica della CPU, il contenuto del Program-Counter viene solo messo in output, come un indirizzo, durante il primo periodo di Timing di un'istruzione. Il contenuto del Data Counter, analogamente, viene messo in output solo durante il secondo periodo di Timing di un'istruzione. Questa CPU più semplice richiederà sei periodi di Timing per eseguire l'istruzione di caricamento del Program-Counter:

COSA DOVREBBE FARE UN'ISTRUZIONE

Vediamo ora in che modo le istruzioni descritte in questo capitolo possono essere rese più semplici o più complesse.

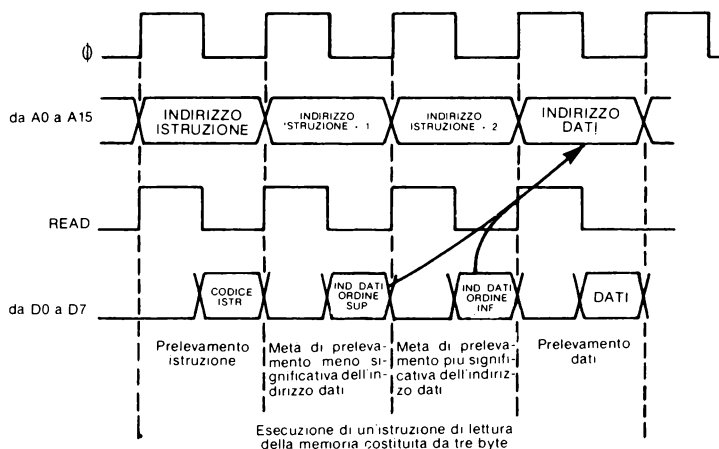
L'istruzione di caricamento del Data Counter carica le due parole di 8 bit che seguono il codice istruzione nel contenuto Data-Counter. Questa istruzione potrebbe essere spezzata in due istruzioni, una per caricare il byte di ordine inferiore del Data-Counter, l'altra per caricare il byte di ordine superiore. Confrontiamo le due forme dell'istruzione di caricamento del Data-Counter.

Istruzione di Caricamento del Data Counter a tre byte

Due istruzioni a due byte per il Caricamento del Data Counter

I segnali e il Timing possibili dell'istruzione di tre byte di caricamento del Data Counter, sono già stati illustrati. Le due istruzioni a 2 byte potrebbero essere eseguite ognuna in due o in quattro periodi di tempo. Con l'esecuzione in due periodi, i segnali e il Timing di ogni istruzione sarebbero i seguenti:

Il Timing e i segnali, con l'esecuzione in quattro periodi di tempo, sarebbero i seguenti:

Spezzando l'istruzione di tre byte di caricamento del Data Counter in due istruzioni di 2 byte, non si semplificano le necessità di logica esterna fatte dalla CPU, ma si rende più semplice l'unità di controllo del microcomputer, come dimostreremo più avanti in questo capitolo quando descriveremo la microprogrammazione.

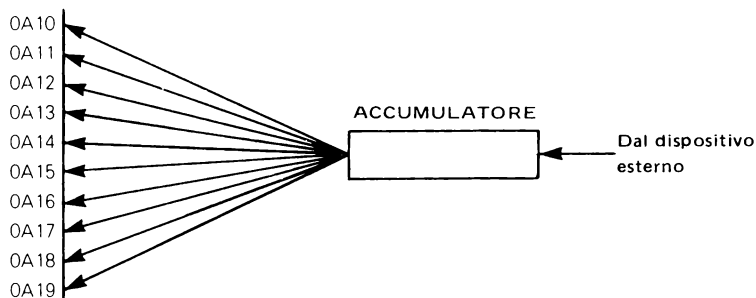
Ora vediamo di combinare le istruzioni di caricamento del Data Counter e di lettura di memoria nel modo seguente:

Come precedentemente illustrato, 41_{16} è il codice istruzione che specifica questa istruzione di lettura in memoria di tre byte; $0A30_{16}$ è l'indirizzo della parola di memoria il cui contenuto deve essere letto nell'accumulatore. Per le istruzioni che specificano l'indirizzo di memoria a cui far riferimento, come questa istruzione di tre byte di lettura di memoria, si dice che usano l'indirizzamento di memoria diretto.

INDIRIZZAMENTO DIRETTO

I segnali e il Timing delle istruzioni di lettura in memoria di tre byte possono assumere varie configurazioni. Ecco la possibilità più compatta.

Per il progettista di un microcomputer, combinare l'istruzione di caricamento del Data Counter con la lettura in memoria — o con qualunque altra istruzione di riferimento alla memoria — è ovvio.

Per il progettista del microcomputer, le virtù automatiche dell'indirizzamento diretto non son così ovvie, per la evidente ragione che le istruzioni di indirizzamento diretto richiedono una logica della unità di controllo più complessa; di questo parleremo più avanti in questo capitolo quando tratteremo la microprogrammazione.

Ma vi è una ragione meno evidente per cui l'indirizzamento diretto non è l'ideale; mentre la maggior parte dei programmi dei minicomputer sono memorizzati nella RAM, la maggior parte dei programmi dei microcomputer sono memorizzati nella ROM; cioò significa che l'indirizzameno diretto può essere usato solo quando l'indirizzo dei dati non cambia.

TABELLE DATI **Consideriamo un'elementare sequenza di istruzioni che riceve dati in input da un dispositivo esterno e poi li memorizza in un certo numero di parole di memoria consecutive:**

Le parole di memoria dei dati della RAM nell'illustrazione precedente costituiscono una "tabella dati". L'indirizzo iniziale $0A10_{16}$ è stato scelto arbitrariamente; andrebbe bene qualunque altro indirizzo.

Ignorando per il momento la questione riguardante il numero di parole dati che devono essere memorizzate nella tabella dati, tale tabella dovrebbe essere formata dalla seguente sequenza di istruzioni:

Un'istruzione diretta di caricamento del Data Counter, memorizzata nelle parole di memoria di programma 0280_{16}, 0281_{16} e 0282_{16} carica l'indirizzo $0A10_{16}$ nel Data Counter; questo è l'indirizzo della prima parola nella tabella dati.

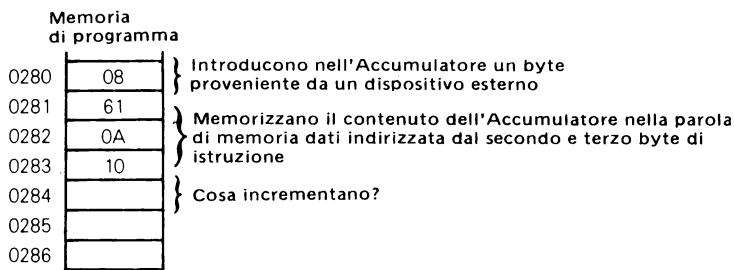
Il codice istruzione 08_{16} nella parola di memoria di programma 0283_{16}, fa sì che un byte dati venga inserito nell'accumulatore da un dispositivo esterno.

Il codice istruzione 60_{16}, nella parola di memoria di programma 0284_{16}, è una semplice istruzione di memorizzazione; essa fa sì che il contenuto dell'accumulatore sia memorizzato nella parola indirizzata dal Data Counter; inizialmente questa era la prima parola della tabella dati, con indirizzo $0A10_{16}$.

LOOP DI PROGRAMMA **Le due istruzioni seguenti, posizionate nelle parole di memoria di programma 0285_{16}, 0286_{16} e 0287_{16}, incrementano il contenuto del Data Counter** (per indirizzare la parola seguente della tabella dati), **poi cambiano il valore del Program Counter in 0283_{16}**; l'esecuzione ritorna ora all'istruzione che porta la parola dati seguente da un dispositivo

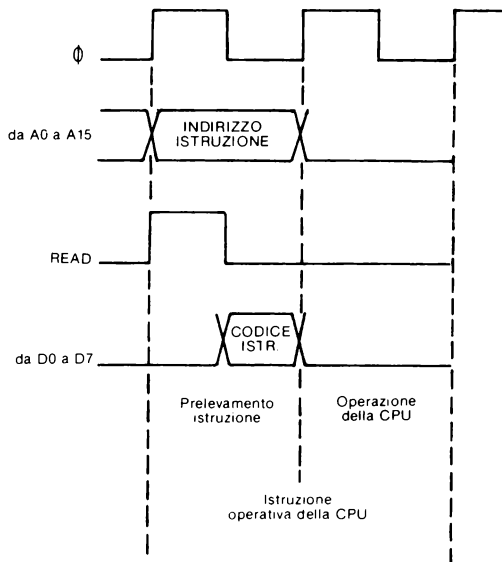
esterno nell'accumulatore. Abbiamo fatto un loop di programma — cioè un gruppo di istruzioni che vengono continuamente rieseguite; ad ogni riesecuzione viene eseguito qualcosa di leggermente diverso, perchè il contenuto del Data Counter viene incrementato ad ogni passaggio.

Quattro istruzioni, che occupano cinque byte di memoria, possono riempire una tabella dati, qualunque sia la lunghezza della tabella stessa!

Usando l'indirizzamento diretto, questo loop non potrebbe essere eseguito. Avremo:

La tabella dati è indirizzata dal secondo e dal terzo byte dell'istruzione di memorizzazione con indirizzamento diretto. Questo indirizzo non può essere incrementato se viene memorizzato nella ROM! I minicomputer hanno una soluzione per questo problema, naturalmente (vedremo qual'è la soluzione nel Capitolo 6), ma la soluzione porta ulteriore complessità ai microcomputer, e la complessità comporta più costi.

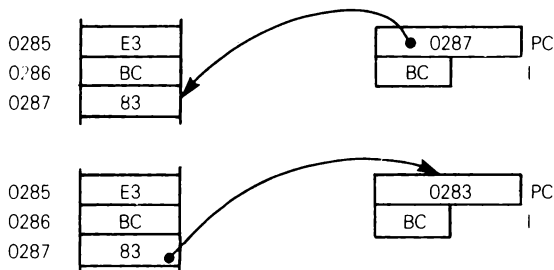
Due delle nuove istruzioni del loop devono ora essere descritte più dettagliatamente.

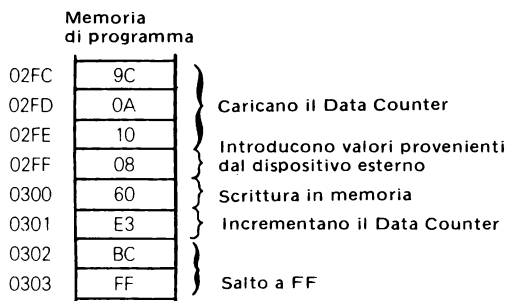
ISTRUZIONI OPERATIVE DELLA CPU

L'istruzione che incrementa il Data Counter ne fa semplicemente aumentare il contenuto del contatore dati di 1; seguendo il prelevamento dell'istruzione, la logica esterna alla CPU è inutile:

ISTRUZIONI DI SALTO E DI BRANCH

L'istruzione nelle parole di memoria di programma 0286_{16} e 0287_{16} cambia il contenuto del Program Counter, e cambia così la sequenza in cui vengono eseguite le istruzioni. Si tratta di un'istruzione di salto o di branch.

Le istruzioni di salto variano l'una dall'altra. Nel loop ne è illustrata una versione a due parole; il contenuto della seconda parola dell'istruzione viene caricato nella metà inferiore del Program Counter, in questo modo:

SALTO ASSOLUTO

SALTI AL LIMITE DELLA PAGINA

Il problema di questo tipo di istruzione di salto è che non funziona se il byte di ordine superiore del Program Counter deve essere incrementato. Per esempio, supponiamo che il loop fosse memorizzato nel modo seguente:

Il salto a FF andrebbe a $03FF_{16}$, non a $02FF_{16}$, perchè il byte di ordine superiore del Program Counter è stato incrementato fra l'istruzione di input e quella di scrittura in memoria.

Vi sono due modi di risolvere questo problema.

Primo, possiamo avere un'istruzione di salto a tre parole che cambia entrambe le metà del Program Counter.

SALTO RELATIVO DI PROGRAMMA

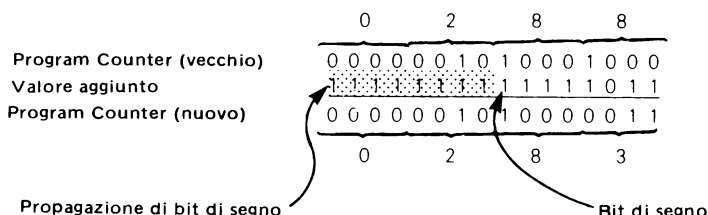
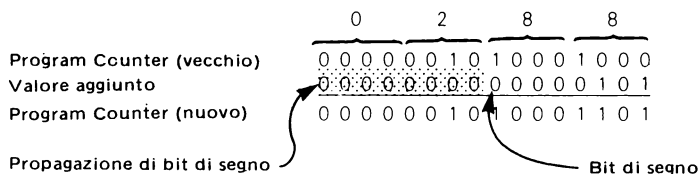
Secondo, possiamo aggiungere il contenuto della seconda parola dell'istruzione di salto al Program Counter, progettando la CPU in modo che interpreti la seconda parola dell'istruzione di salto come un numero binario con segno. In riferimento al loop, do-

po che l'istruzione di salto è stata eseguita, il Program Counter conterrebbe normalmente 0288_{16}; per cambiare questo valore in 0283_{16}, bisogna sottrarre 5. Il complemento a due di 5 è:

1 1 1 1 1 0 1 1

o FB_{16}. Questo è il valore che dovrebbe essere memorizzato nella parola di memoria di programma 0287_{16}.

PROPAGAZIONE DEL SEGNO

Aggiungere un valore di 8 bit al contenuto a 16 bit del Program-Counter usando l'aritmetica binaria con segno, non è un problema; la logica della CPU propaga semplicemente il bit del segno attraverso la metà di ordine superiore del valore che va aggiunto al Program Counter. In questo caso abbiamo:

Per aggiungere 5 al contenuto del Program Counter si procederebbe in questo modo:

Questo è un salto relativo di programma.

LA MICROPROGRAMMAZIONE E L'UNITA' DI CONTROLLO

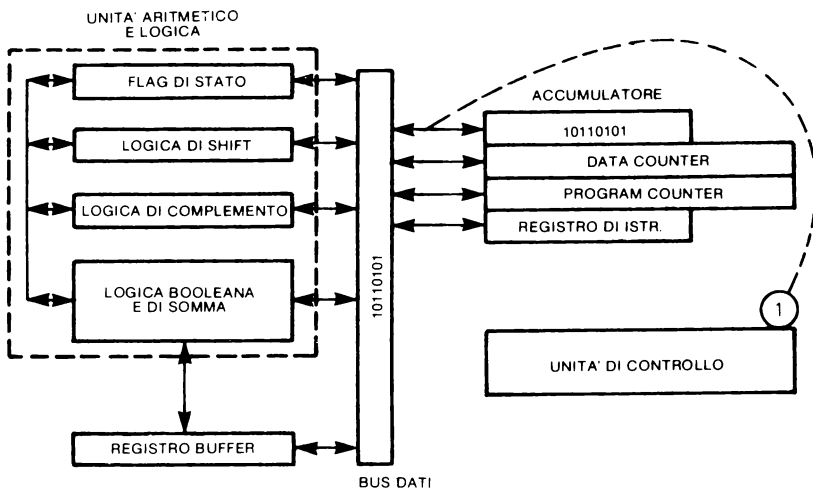
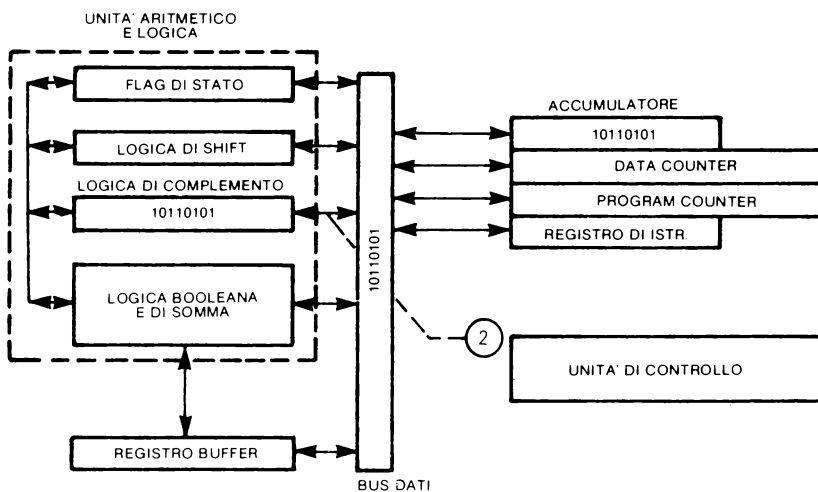
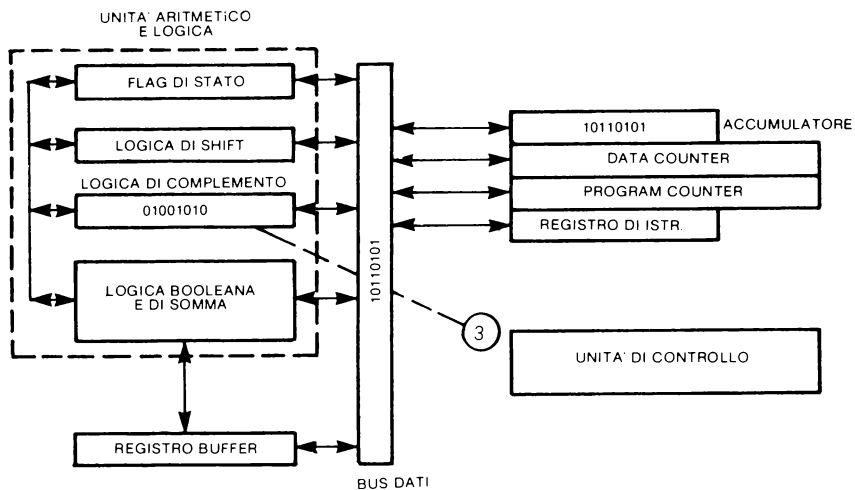
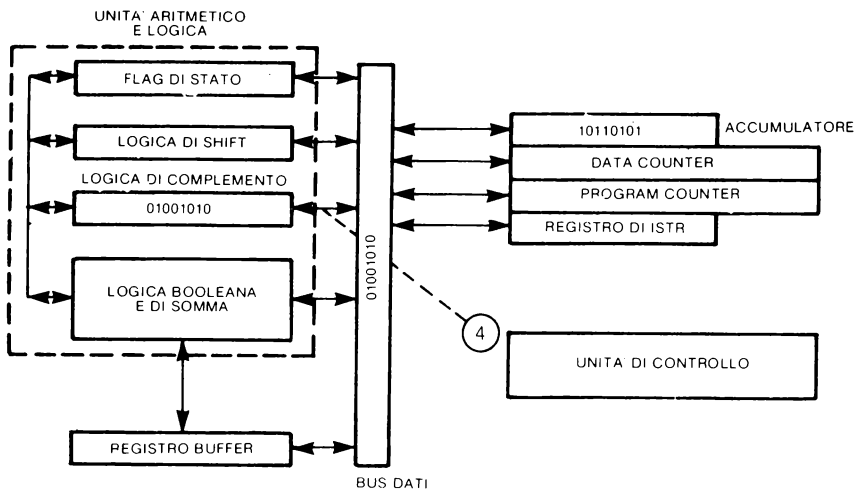
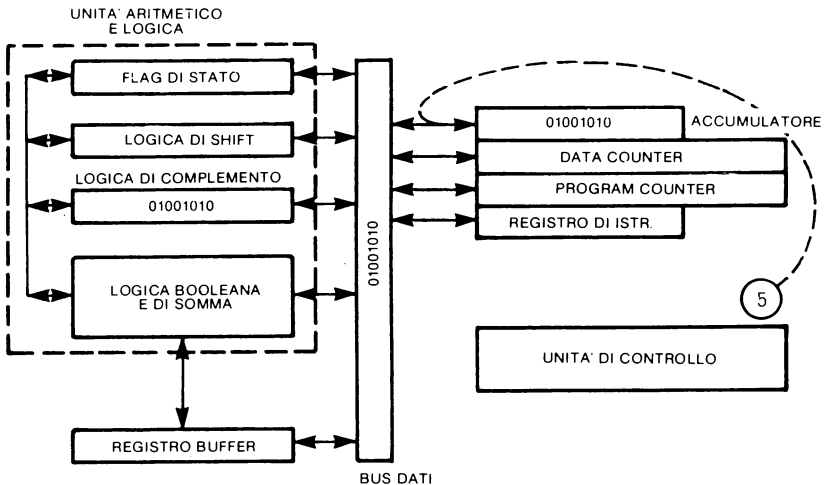
Vediamo ora in che modo l'unità di controllo decodifica i codici istruzione.

La CPU di un microcomputer può essere illustrata in modo funzionale come nella Figura 4-1, ma in realtà la CPU consiste di un numero di elementi logici, attivati da sequenze di segnali di abilitazione.

Il complementatore, ad esempio, è in grado, in qualunque momento, di complementare i contenuti di otto latch dati all'interno della logica dei circuiti di complemento. Un solo segnale di abilitazione, proveniente dall'unità di controllo, attiverà questa sequenza logica.

Comunque, il fatto di complementare otto latch dati all'interno del complementatore non è di per se stessa di alcuna utilità. In realtà si vuole complementare il contenuto dello accumulatore, e ciò significa spostare il contenuto nel complementatore, poi, dopo aver abilitato la logica del complementatore, rimandarne il risultato nell'accumulatore.

Per complementare il contenuto dell'accumulatore occorrono cinque passi:

1) Spostare i contenuti dell'accumuiatore nel bus dati.

2) Spostare il contenuto del bus dati nel complementatore.

3) Attivare la logica del complementatore:

4) Spostare il contenuto del complementatore nel bus dati:

5) Spostare il contenuto del bus dati nell'accumulatore:

MICROISTRUZIONI E MACROISTRUZIONI

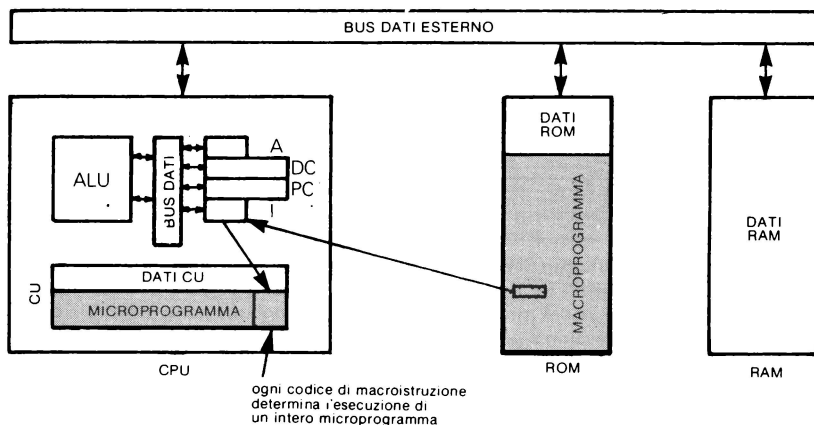
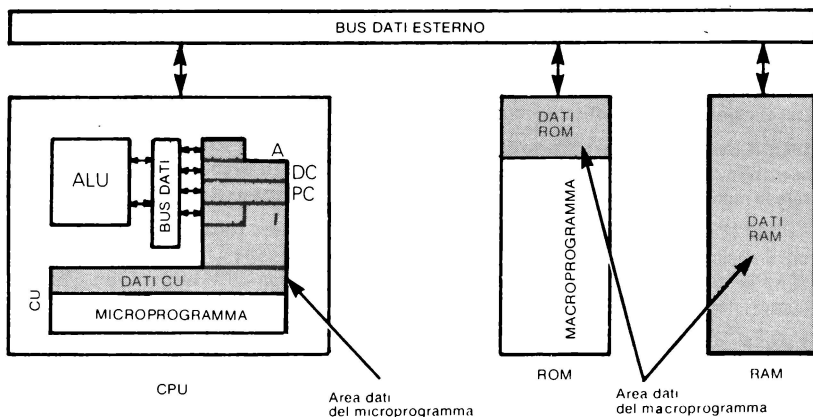
Ognuno di questi cinque passi è una microistruzione. Ogni microistruzione è abilitata da un segnale proveniente dall'unità di controllo. Mettendo in output la sequenza di segnali di controllo appropriata, **l'unità di controllo può mettere in sequenza un numero qualunque di microistruzione, per creare una macroistruzione, che è la risposta accettata della CPU ad un codice istruzione in linguaggio assemblatore.**

MICROPROGRAMMI

Per complementare il contenuto dell'accumulatore, l'unità di controllo deve contenere i cinque codici binari, oguno dei quali genera uno o più segnali di controllo appropriati. Questa sequenza di codici binari all'interno dell'unità di controllo costituisce un microprogramma. La microprogrammazione consiste quindi nel generare la sequenza di codici binari che sono memorizzati all'interno dell'unità di controllo.

Vi è uno stretto parallelismo fra la microprogrammazione e la programmazione in linguaggio assemblatore.

Un microprogramma viene memorizzato nell'unità di controllo come una sequenza di cifre binarie. Un programma in linguaggio assemblatore è memorizzato come una sequenza di cifre binarie, di solito in una memoria ROM. Il programma in linguaggio

assemblatore è considerato un macroprogramma. Ogni codice del macroprogramma dà inizio all'esecuzione di un intero microprogramma, come è stato memorizzato nell'unità di controllo:

Un microprogramma memorizzato nell'unità di controllo ha una memoria dati, che consiste dei registri della CPU, più una memorizzazione dati interna all'unità di controllo. Un macroprogramma ha un'area dati, che consiste di memoria ROM per i dati costanti più una memoria RAM per i dati variabili:

Le singole istruzioni di un microprogramma implementano una piccola sequenza logica all'interno della logica della CPU. Le singole istruzioni di un macroprogramma fanno sì che un intero microprogramma venga eseguito, implementando così tutta una sequenza di operazioni all'interno della CPU.

COMPLESSITA' DELLE MACROISTRUZIONI

La complessità delle operazioni associate a una qualunque macroistruzione è funzione diretta della lunghezza del microprogramma di cui la macroistruzione fa partire l'esecuzione. Non vi sono punti particolari o livelli logici ai quali il microprogrammatore deve terminare il microprogramma che sarà eseguito in risposta a qualunque codice macroistruzioni. Naturalmente, i microprogrammi complessi richiedono una grossa unità di controllo. Un microcomputer semplice può avere una piccola unità di controllo e quindi gli si possono far eseguire solo macroistruzioni molto semplici. Alcuni grossi computer non hanno linguaggio assemblatore, ma, in risposta ad un solo codice di macroistruzione, eseguono complesse sequenze di eventi che coinvolgono la logica dell'intero sistema.

MICROCOMPUTER MICROPROGRAMMABILI

L'unità di controllo di ogni microcomputer in realtà non è altro che un microprogramma. Se voi, come utente, avete la possibilità di creare o modificare il microprogramma all'interno dell'unità di controllo, si dice allora che il microcomputer è "microprogrammabile". Se il microprogramma dell'unità di controllo è progettato dal progettista di logica del microcomputer, e diventa poi una parte inalterabile della CPU, il microcomputer non è programmabile.

In questo libro descriveremo queste due classi separate e distinte di microcomputer oggi prodotti:

1) **Microcomputer basato su un microprocessore che vi dà accesso all'unità centrale, ma non all'unità di controllo.** Voi ordinate in sequenza la logica della CPU usando macroistruzioni, che tutte insieme costituiscono un set di istruzioni in linguaggio assemblatore. Non potete microprogrammare questa classe di microcomputer; ma una comprensione di base della microprogrammazione vi aiuterà comunque a capire i fattori che ogni progettista di microcomputer deve valutare quando definisce un set di istruzioni.

2) **"Microprocessor slice" o microcomputer basato sulla macrologica.** Qui vi trovate di fronte agli elementi costitutivi la CPU, che dovete collegare per formare un microprogramma.

MICROCOMPUTER BASATI SUI MICROPROCESSORI

Descriveremo prima i microcomputer basati sui microprocessori.

Identifichiamo un set arbitrario di segnali di controllo, come illustra la Fig.4-1. L'unità di controllo del nostro microcomputer genererà questi segnali di controllo per implementare le macroistruzioni. Le Tabelle 4-1, 4-2 e 4-3 descrivono questi segnali di controllo.

Se confrontate con l'ingegnosità degli attuali codici di microistruzioni di chip slice, tali Tabelle rappresentano un'organizzazione della logica della CPU che ha qualcosa di semplicistico e di rigido. Ciononostante, esse renderanno l'architettura del chip slice più comprensibile chiarendo gli scopi a cui il progettista di logica del chip deve attenersi.

I segnali di controllo descritti nelle Tabelle suddette non permettono all'unità di controllo di eseguire le operazioni necessarie per dare un supporto alle istruzioni in linguaggio assemblatore. Ad esempio, non abbiamo detto nulla su come verranno generati i segnali di controllo di READ e WRITE, o di come verranno trattati i quattro latch di stato C (Carry), O (Overflow), S (Segno) e Z (Zero). Un sistema primitivo di affrontare questi problemi è bloccare le microistruzioni che tentano di settare sia C0 che C1, che è una condizione impossibile, dato che tende a spostare simultaneamente da e verso il bus. **Se C0 e C1 sono entrambi 1, saranno messi in output 0, e i segnali rimanenti, da C2 a C8, verranno intepretati come definenti una delle seguenti cinque diverse classi di operazioni interne all'unità di controllo:**

1) Se C2 . . . . C8 sono tutti 0, i latch Z, S, O e C, nella ALU registreranno le loro condizioni nel buffer CU DATA.

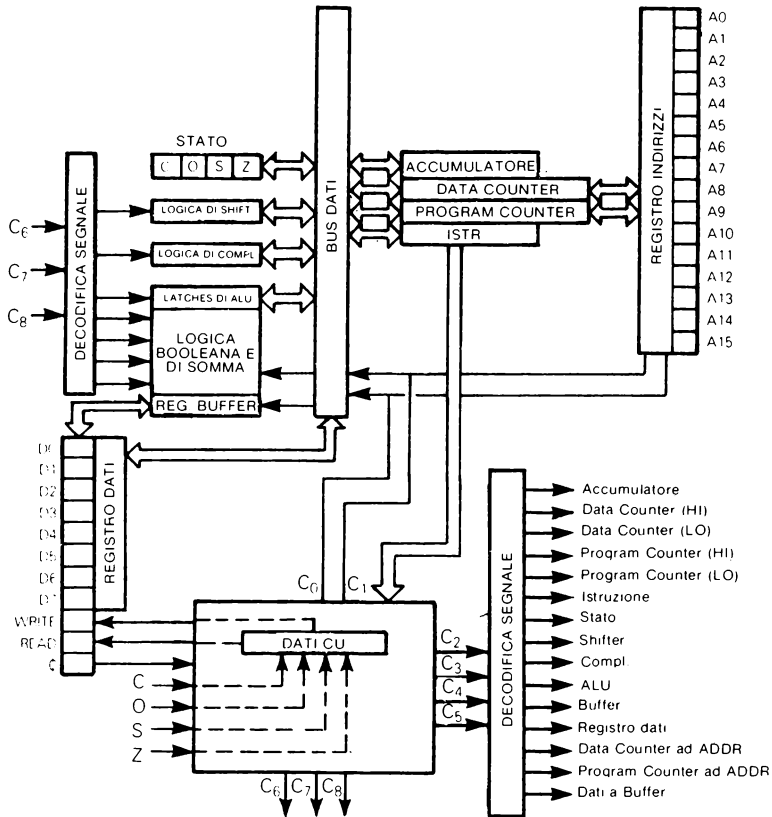
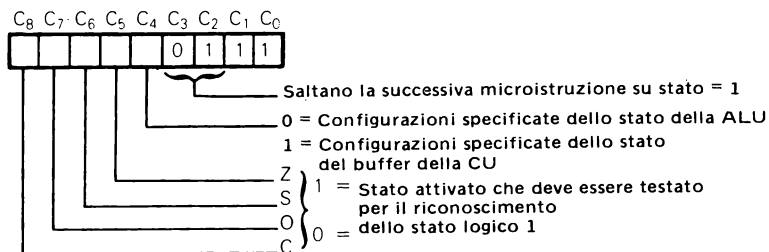
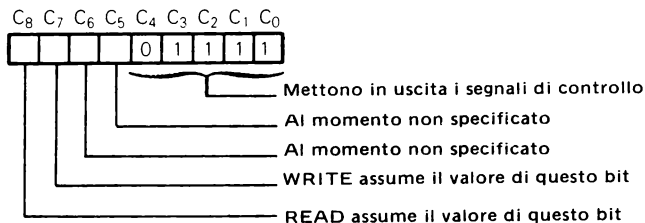



Figura 4-2. Segnali dell'Unità di Controllo per un microcomputer semplice

2) Se C2 e C3 sono a 1 e 0, allora C5, C6, C7 e C8 verranno interpretati come corrispondenti agli stati Z, S, O e C, rispettivamente. Se C4 è 0, ci si riferisce alle condizioni di stato nella ALU; se C4 è 1, si fa riferimento alle condizioni di stato memorizzate nel buffer dati. C5, C6, C7 e C8 saranno tutti testati per cercare il valore 1. Se si trova un 1, verrà controllato lo stato corrispondente. Se lo stato corrispondente ha valore 1, verrà saltata la microistruzione seguente. Questo uso dei nove controlli da C_0 a C_8 può essere illustrato nel modo seguente:

3) Se C2 e C3 sono 0 e 1, verrà ripetuta la logica della condizione 2 prima descritta; comunque, i flag di stato corrispondenti saranno controllati per i valori di 0 che è la condizione che obbliga a saltare l'istruzione seguente.

4) Se C2, C3 e C4 sono 1, 1 e 0, rispettivamente, C5, C6, C7 e C8 specificano lo stato di quattro segnali di controllo che l'unità di controllo può mettere in output ai pin del chip. Abbiamo descritto solo due segnali di controllo fino ad ora: READ e WRITE. Supporremo che C8 specifichi la condizione di READ e che C7 specifichi la condizione di WRITE. L'uso di nove controlli da C_0 a C_8 può essere illustrato in questo modo:

5) Quando C2, C3 e C4, sono tutti 1, C5 . . . . . C8 saranno decodificati internamente per determinare una delle 16 operazioni logiche interne all'unità di controllo. Non tenteremo di definire che cosa potrebbero essere queste operazioni.

Creeremo ora dei microprogrammi. Incominciamo in maniera semplice, creando il microprogramma di prelevamento di una istruzione. Ricordate che l'esecuzione di tutte le istruzioni inizia con il prelevamento dell'istruzione stessa; perciò, il programma di prelevamento deve precedere tutti i microprogrammi che implementano l'esecuzione di un'istruzione. Questo microprogramma è mostrato nella Tabella 4-4.

Prima di analizzare il programma, microistruzione per microistruzione, dobbiamo fare alcuni commenti di carattere generale.

LUNGHEZZA IN BIT DELLE MACROISTRUZIONI

Ogni microistruzione diventa di 9 cifre binarie all'interno dell'unità di controllo. L'unità di 8 bit (o byte) è stata selezionata come la lunghezza di parole per il microcomputer perchè questa lunghezza di parola è utile quando si rappresentano caratteri e dati numerici, oltre che nella rappresentazione di codici istruzione. Il microprogramma dell'unità di controllo non rappresenta dati numerici o codici istruzioni; perciò, la lunghezza in bit della microistruzione è scelta della

Tabella 4-1. Segnali dell'Unità di controllo

SEGNALE	FUNZIONE	
C0, C1	C0=0, C1=0	Nessun dato viene mosso da o per Bus dati o Registro di indirizzamento
	C0=1, C1=0	I dati vengono mossi verso Bus dati o Registro di indirizzamento
	C0=0, C1=1	I dati vengono prelevati da Bus dati o Registro di indirizzamento
	C0=1, C1=1	La microistruzione viene bloccata all'interno della unità di controllo
C2, C3 C4, C5	Quando C0=1, C1=0 o C0=0, C1=1 questi quattro segnali vengono decodificati per selezionare un movimento di dati, come viene specificato nella Tabella 4-2.	
C6, C7, C8	Questi tre segnali vengono decodificati per selezionare operazioni della ALU come specificato nella Tabella 4-3.	
WRITE, READ	Connessioni dirette tra i pin di input e due bit dati della CU.	
Φ	Segnale di clock messo in input alla CU.	
C, O, S, Z	Quattro bit di stato direttamente collegati a quattro bit dati della CU.	

Tabella 4-2. Selezione del flusso dei dati quando C0=1 e C1=1

C_5	C_4	C_3	C_2	FUNZIONE
0	0	0	0	Accumulatore ⟷ Bus Dati
1	0	0	0	Byte più significativo del Data Counter ⟷ Bus Dati
0	1	0	0	Byte meno significativo del Data Counter ⟷ Bus Dati
1	1	0	0	Byte più significativo del Program Counter ⟷ Bus Dati
0	0	1	0	Byte meno significativo del Program Counter ⟷ Bus Dati
1	0	1	0	Registro delle istruzioni ⟷ Bus Dati
0	1	1	0	Registro di stato ⟷ Bus Dati
1	1	1	0	Shifter ⟷ Bus Dati
0	0	0	1	Complementatore ⟷ Bus Dati
1	0	0	1	Latches dell'ALU ⟷ Bus Dati
0	1	0	1	Buffer dell'ALU ⟷ Bus Dati
1	1	0	1	Registro Dati ⟷ Bus Dati
0	0	1	1	Data Counter ⟷ Registro di indirizzamento
1	0	1	1	Program Counter ⟷ Registro di indirizzamento
0	1	1	1	Registro Dati ⟷ Registro Buffer
1	1	1	1	Non usata

Tabella 4-3. Segnali di selezione dell'ALU

C_8	C_7	C_6	FUNZIONE
0	0	0	Selezione della logica di Shift (rotazione)
1	0	0	Selezione della logica di Complementazione
0	1	0	Selezione della logica di Addizione *
1	1	0	Selezione della logica dell'AND*
0	0	1	Selezione della logica dell'OR*
1	0	1	Selezione della logica dello XOR*
0	1	1	Incrementa i latches dell'ALU
1	1	1	Nessuna operazione dell'ALU

* L'operazione viene effettuata sui contenuti dei latches ALU e del registro di Buffer. Il risultato compare nei latches ALU.

lunghezza che occorre al microcomputer — in questo caso nove bit. Dato che vi sono 15 microistruzioni nel microprogramma di prelevamento dell'istruzione, saranno necessarie in totale, per contenere il programma suddetto, 135 cifre binarie, disposte in una matrice di 9 x 15.

Notate anche che viene eseguita una sequenza di 15 microistruzioni durante il prelevamento di una istruzione, e che deve essere eseguito durante un solo periodo di clock Ø. Perciò l'unità di controllo dividerà internamente il segnale di clock in 16 parti. In altre parole, se il clock ha un periodo di un microsecondo, ogni microistruzione deve essere eseguita entro 62,5 nanosecondi. Dato che i chip della CPU consistono mediamente di logica n-MOS o p-MOS strettamente impaccata, questo tempo è ragionevole.

Ora consideriamo dettagliatamente i 15 passi del microprogramma di prelevamento di una istruzione.

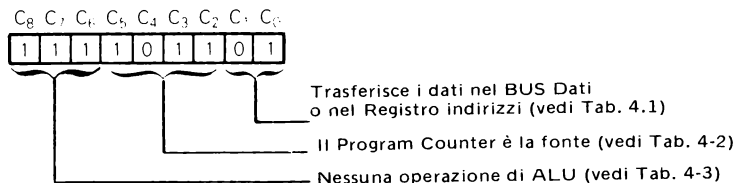
I primi due bit del codice di 9 bit della prima microistruzione rappresentano C0 e C1, sono settati a 1 e 0, rispettivamente; quindi indicano che i dati saranno spostati nel Bus dati o nel Registro di indirizzamento (vedi Tabella 4-1). I quattro bit seguenti sono a 1 1 0 1; essi specificano che è il contenuto del Program-Counter che deve essere spostato nel Registro di indirizzamento (vedi Tabella 4-2). Dato che non deve aver luogo nessuna operazione simultanea nella ALU, gli ultimi tre bit sono tutti settati a 1 (vedi Tabella 4-3). La creazione di questa microistruzione è così illustrata:

Se avete delle difficoltà a capire il funzionamento della prima microistruzione, dovreste studiarne dettagliatamente alcune altre, per vedere come funzionano, tenendo anche conto delle informazioni date nelle Tabelle 4-1, 4-2 e 4-3.

Tabella 4-4. Microprogramma di prelevamento di una microistruzione

Numero della istruzione	CODICE DELLA MICROISTRUZIONE									FUNZIONI
	C_8	C_7	C_6	C_5	C_4	C_3	C_2	C_1	C_0	
1	1	1	1	1	0	1	1	0	1	Sposta il Program Counter nel Registro di indirizzamento
2	1	0	0	0	0	1	1	1	1	Mette a 1 (vero) il segnale READ e a 0 (falso) il segnale WRITE
3	1	1	1	0	0	1	0	0	1	Manda il byte meno significativo del Program Counter nel Bus Dati
4	1	1	1	1	0	0	1	1	0	Sposta il contenuto del Bus Dati nei latches dell'ALU
5	0	1	1	0	0	0	0	0	0	Incrementa i latches della ALU
6	1	1	1	1	0	0	1	0	1	Sposta il contenuto dei latches della ALU nel Bus Dati
7	1	1	1	0	0	1	0	1	0	Sposta il contenuto del Bus Dati nel byte di ordine inferiore del Program Counter
8	1	1	1	1	1	0	0	0	1	Sposta il contenuto del byte di ordine superiore del Program Counter nel Bus Dati
9	1	1	1	1	0	0	1	1	0	Sposta il contenuto del Bus Dati nei latches dell'ALU
10	1	0	0	0	0	1	0	1	1	Salta la prossima microistruzione se lo stato carry è = 0
11	0	1	1	0	0	0	0	0	0	Incrementa i latches dell'ALU
12	1	1	1	1	0	0	1	0	1	Sposta il contenuto dei latches della ALU nel Bus Dati
13	1	1	1	1	1	0	0	1	0	Sposta il contenuto del Bus Dati nel byte di ordine superiore del Program Counter
14	1	1	1	1	1	0	1	0	1	Sposta il contenuto del registro dati nel Bus Dati
15	1	1	1	1	0	1	0	1	0	Sposta il contenuto del Bus Dati nel registro istruzioni

La microistruzione 1 sposta il contenuto del Program Counter nel registro di indirizzamento, facendo così apparire il contenuto di 16 bit del Program Counter ai 16 pin d'indirizzamento. L'istruzione 2 setta il segnale di controllo di WRITE a 0 (falso), e il segnale di controllo di READ a 1 (vero); ciò dice alla logica esterna che i pin da A0 ad A15 forniscono l'indirizzo di una parola di memoria esterna, i cui contenuti devono essere posti dal pin D0 al pin D7. La logica esterna ha 687,5 nanosecondi, cioè il tempo che occorre per eseguire le microistruzioni da 3 a 13; per prelevare i dati richiesti.

LO STATO NEI MICROPROGRAMMI

Le microistruzioni da 3 a 13 incrementano il contenuto del Program Counter, come richiesto durante ogni prelevamento d'istruzione. Dato che il Program Counter è di 16 bit, mentre la logica all'interno della CPU è solo di 8 bit, deve essere incrementato di due passi. Le istruzioni da 3 a 7 ne incrementano la metà di ordine inferiore. Se risulta che questo incremento ha fatto andare a 1 lo stato Carry (solo nella ALU), anche la metà superiore del Program Counter deve essere incrementata. Se lo stato Carry non è settato, la metà superiore del Program Counter deve restare inalterata. Le microistruzioni da 9 a 13 trattano la metà superiore del Program Counter. Queste microistruzioni sono parallele alle microistruzioni da 3 a 7; comunque, la microistruzione 10 specifica che, se lo stato di Carry (nella ALU) è 0, bisogna saltare la microistruzione 11, che esegue l'incremento del contenuto dei latch della ALU. Così la metà superiore del Program Counter è incrementata solo se lo stato Carry è stato settato quando la metà inferiore è stata incrementata.

Notate che la logica dell'unità di controllo deve essere molto precisa a proposito di quando deve registrare gli stati nel suo buffer CU DATA e quando non lo deve fare. L'uso dello stato Carry (C) come mezzo per controllare l'incremento del Program Counter, è valido solo se lo stato Carry non è permanentemente registrato nell'unità di controllo. In altre parole, l'unità di controllo può riferire sui latch di stato nella ALU in qualunque momento. Le istruzioni in linguaggio assemblatore si riferiscono sugli stati memorizzati nel buffer CU DATA, mai sugli stati latch della ALU. Il codice di microistruzione 00000011 deve essere eseguito dall'unità di controllo se gli stati nei latch della ALU devono essere conservati nel buffer CU DATA.

MICROPROGRAMMA DI COMPLEMENTO

Ora prendiamo in esame i cinque passi che occorrono per complementare il contenuto dell'accumulatore. Se, durante il 15° passo del microprogramma di prelevamento dell'istruzione, il codice caricamento nel registro istruzioni è un codice istruzione per complementare l'accumulatore, la logica dell'unità di controllo salterà al microprogramma mostrato nella Tabella 4-5.

Tabella 4-5. Programma di complemento dell'accumulatore

Numero della istruzione	CODICE DELLA MICROISTRUZIONE									FUNZIONE
	C_8	C_7	C_6	C_5	C_4	C_3	C_2	C_1	C_0	
1	1	1	1	0	0	0	0	0	1	Sposta l'accumulatore nel Bus Dati
2	1	1	1	0	0	0	1	1	0	Sposta il Bus Dati nel Complementatore.
3	1	0	0	0	0	0	0	0	0	Esegue la logica di complemento
4	1	1	1	0	0	0	1	0	1	Sposta il complementatore nel Bus Dati
5	1	1	1	0	0	0	0	1	0	Sposta il Bus Dati nell'accumulatore

Nelle Tabelle 4-6, 4-7 e 4-8 sono illustrati i tre modi con cui è possibile caricare i dati nell'accumulatore.

Per complementare l'accumulatore, bisogna eseguire un microprogramma di 45 bit. Anche se queste cinque microistruzioni possono essere eseguite in 312,5 nanosecondi, la sincronizzazione del sistema richiede che per l'esecuzione dell'istruzione venga usato almeno un periodo di clock Φ; perciò, il tempo rimanente andrà sprecato.

MICROPROGRAMMI CON ISTRUZIONI IN LINGUAGGIO ASSEMBLATORE

Consideriamo ora i cambiamenti che avvengono a seconda se si hanno sequenze di istruzioni semplici o complesse. Facendo riferimento al programma di addizione binaria descritto in precedenza in questo Capitolo, ricorderete che una parola di dati può essere caricata dalla memoria nell'accumulatore in uno dei seguenti modi:

1) Originare due istruzioni separate, ognuna delle quali carica metà del Data Counter con metà dell'indirizzo di memoria dei dati per la parola di memoria il cui contenuto deve essere caricato nell'accumulatore. Poi eseguire una terza istruzione per caricare il contenuto nell'accumulatore.
2) Usare un'istruzione per caricare nel Data Counter tutto l'indirizzo di memoria della parola il cui contenuto deve essere letto nell'accumulatore. Poi eseguire una seconda istruzione per spostarne il contenuto nell'accumulatore.
3) Avere una sola istruzione a indirizzamento diretto che carica l'indirizzo della parola nel Data Counter e poi ne carica il contenuto nell'accumulatore.

Tabella 4-6. Tre istruzioni di lettura in memoria

Numero della istruzione	CODICE DELLA MICROISTRUZIONE									FUNZIONE
	C_8	C_7	C_6	C_5	C_4	C_3	C_2	C_1	C_0	
1										Ripete le microistruzioni da 1 a 14 del prelevamento dell'istruzione (Tabella 4-4)
— -										
——										
14										
15	1	1	1	0	1	0	0	1	0	Sposta il Bus Dati nel byte meno significativo del Data Counter

(A) Carica il byte meno significativo del Data Counter

Numero della istruzione	CODICE DELLA MICROISTRUZIONE									FUNZIONE
	C_8	C_7	C_6	C_5	C_4	C_3	C_2	C_1	C_0	
1										Ripete le microistruzioni da 1 a 14 del prelevamento dell'istruzione (Tabella 4-4)
--										
--										
--										
14										
15	1	1	1	1	0	0	0	1	0	Sposta il Bus Dati nel byte più significativo del Data Counter

(B) Carica il byte più significativo del Data Counter

Numero della istruzione	CODICE DELLA MICROISTRUZIONE									FUNZIONE
	C_8	C_7	C_6	C_5	C_4	C_3	C_2	C_1	C_0	
1	1	1	1	0	0	1	1	0	1	Sposta il Data Counter nel registro di indirizzamento
2	1	0	0	0	1	0	1	1	1	Mette a 1 (vero) il segnale READ e a 0 (falso) il segnale WRITE
3	1	1	1	0	0	0	0	0	0	Vengono incluse 12 operazioni nulle per dare tempo alla logica esterna di prelevare i dati
--										
--										
--										
14	1	1	1	0	0	0	0	0	0	
15	1	1	1	1	1	0	1	0	1	Sposta il registro dati nel Bus Dati
16	1	1	1	0	0	0	0	1	0	Sposta il Bus Dati nell'accumulatore

(C) Legge il contenuto delle parole di memoria indirizzate nell'accumulatore

Tabella 4-7. Un'istruzione da caricare nell'indirizzo a 16 bit nel Data Counter

Numero della istruzione	CODICE DELLA MICROISTRUZIONE									FUNZIONE
	C_8	C_7	C_6	C_5	C_4	C_3	C_2	C_1	C_0	
1										Ripete le microistruzioni da 1 a 14 di prelevamento dell'istruzione (Tabella 4-4)
--										
--										
--										
14										
15	1	1	1	0	1	0	0	1	0	Sposta il Bus dati nel byte di ordine inferiore del Data Counter
--										
--										
--										
16	1	1	1	0	0	0	0	0	0	Nessunaoperazione per perdere tempo
17										
--										
--										
--										
30										
31	1	1	1	1	0	0	0	1	0	Sposta il Bus dati nel byte più significativo del Data Counter

La Tabella 4-6 (C) è il secondo passo per la tabella 4-7.

Un'occhiata alle Tabelle 4-6, 4-7 e 4-8 mostra che i microprogrammi avranno moltissime sequenze di microistruzioni duplicate. La primissima cosa che un progettista di microcomputer farà è tentare di eliminare questa duplicazione riusando le sequenze di microistruzioni di cui si ha spesso bisogno.

Inoltre, un progettista di microcomputer dovrà sviluppare dei mezzi molto semplici per dare alla logica esterna il tempo di rispondere ad una richiesta di READ, invece di inserire 12 microistruzioni di "no operation", usando fino anche 108 bit della memoria dell'unità di controllo.

Queste complicazioni hanno costretto i primi progettisti di microcomputer a fare delle istruzioni semplici in linguaggio assemblatore. Vi sono molti modi in cui si possono riusare le sequenze dei microprogrammi, nonchè implementare i ritardi di tempo; abbiamo lasciato 16 microistruzioni libere proprio per questo tipo di operazioni. La memoria dell'unità di controllo viene sempre più riempita per risolvere queste complicazioni, e più complicazioni vi sono all'interno di una sola istruzione, più complessa diventerà questa logica extra dell'unità di controllo.

Tabella 4-8. Istruzione singola, con indirizzamento diretto, di lettura in memoria

Numero della istruzione	CODICE DELLA MICROISTRUZIONE									FUNZIONE
	C_8	C_7	C_6	C_5	C_4	C_3	C_2	C_1	C_0	
1										Ripete le microistruzioni da 1 a 14
--										del prelevamento dell'istruzione (Ta-
--										bella 4-4)
--										
14										
15	1	1	1	0	1	0	0	1	0	Sposta il Bus dati nel byte meno si-
--										gnificativo del Data Counter
--										
--										
16	1	1	1	0	0	0	0	0	0	Nessuna operazione per perdere tempo
17										Ripete le microistruzioni da 1 a 14
--										del prelevamento dell'istruzione (Ta-
--										bella 4-4)
--										
30										
31	1	1	1	1	0	0	0	1	0	Sposta il Bus dati nel byte più signi-
--										ficativo del Data Counter
--										
--										
32	1	1	1	0	0	0	0	0	0	Nessunaoperazione per perdere tempo
33	1	1	1	0	0	1	1	0	1	Sposta il Data Counter nel registro di indirizzamento
34	1	0	0	0	1	0	1	1	1	Mette a 1 (vero) il segnale READ e a
--										0 (falso) WRITE
--										
--										
35	1	1	1	0	0	0	0	0	0	Vengono incluse 12 operazioni nulle
--										per dare tempo alla logica esterna di
--										prendere i dati
--										
46	1	1	1	0	0	0	0	0	0	
47	1	1	1	1	1	0	1	0	1	Sposta il registro dati nel Bus dati
48	1	1	1	0	0	0	0	1	0	Sposta il bus dati nell'accumulatore

MICROCOMPUTER "CHIP SLICE"

Supponiamo che un microcomputer basato su di un microprocessore non soddisfi i vostri bisogni: ciò accadrà spesso perchè magari il microprocessore non è abbastanza veloce. A questo punto siete un candidato ai microcomputer "chip slice", o basati sulla "macrologica", che vi permettono di progettare e costruire la vostra CPU, con qualunque tipo di architettura (senza limiti), e qualunque, o nessuno, set di istruzioni in linguaggio assemblatore.

Prima di esaminare in che cosa deve consistere un prodotto "chip slice", facciamo, una precisazione. L'argomento riguardante i prodotti chip slice è un po' come una tangente nel contesto dei prodotti discussi in questo libro.

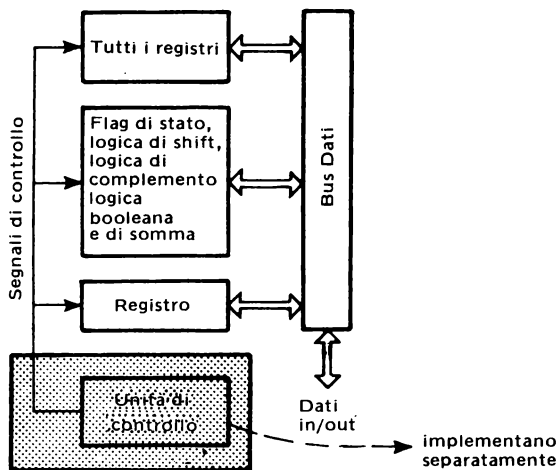
Fin qui abbiamo descritto i microprocessori — la logica della CPU che sarà implementata su di un solo chip, o che può essere parte di un solo chip. Il Capitolo 5 descrive la logica addizionale che fa da supporto ai microprocessori basati sui microcomputer. In confronto ai microprocessori, i prodotti chip slice prendono una direzione filosofica completamente diversa:
Si costruiscono chip con meno logica, dove ogni chip fornisce uno dei blocchi fondamentali di qualunque CPU; allora si giustifica meno logica — e quindi più chip — con un aumento delle prestazioni.

Potreste quindi usare dei chip slice per costruire l'equivalente di qualunque microprocessore. Avreste allora un prodotto con forse dieci chip al posto di uno, ma esso eseguirebbe istruzioni ad una velocità dieci volte maggiore.
Il fatto di descrivere i chip slice alla fine del Capitolo 4, ha senso per il fatto che essi sono essenzialmente blocchi fondamentali della CPU.

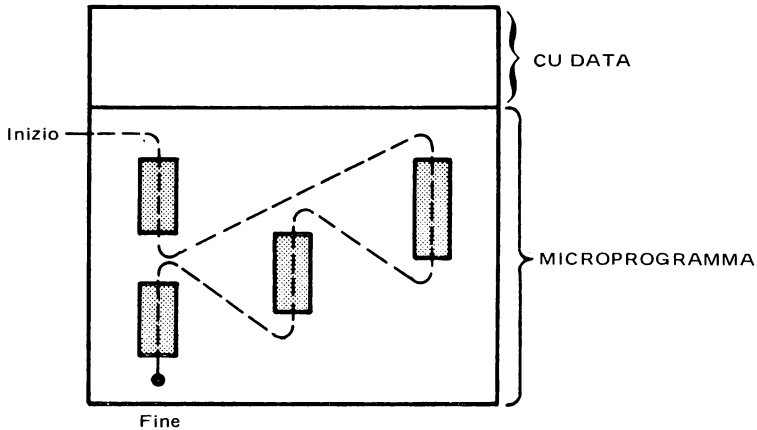
Questa è la teoria in base alla quale scegliamo di descrivere i chip slice; comunque quando avrete finito di leggere il Capitolo 5, vi renderete conto che i chip slice potrebbero essere usati altrettanto bene per costruire l'equivalente di qualunque dispositivo logico di supporto, escluse la ROM e la RAM.

LA FILOSOFIA DEI CHIP SLICE

Se è necessario costruire i blocchi fondamentali della CPU, come dovrebbe essere divisa la sua logica per far sì che i pezzi risultanti siano universali?

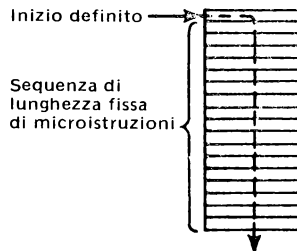
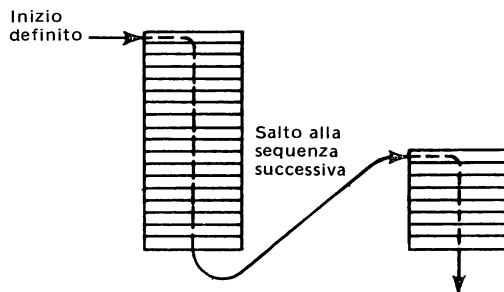
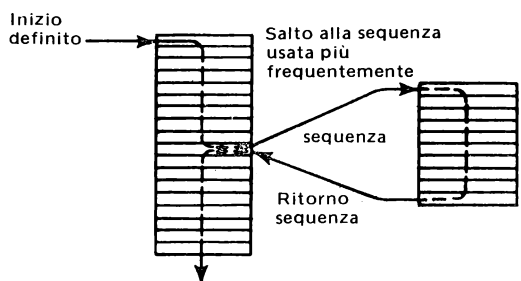
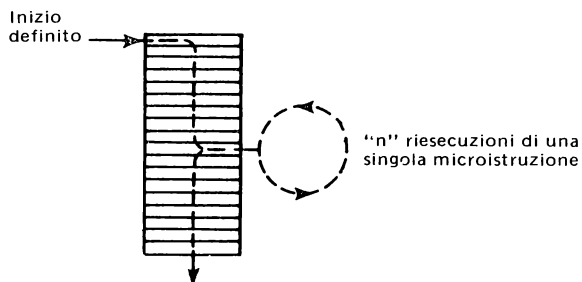
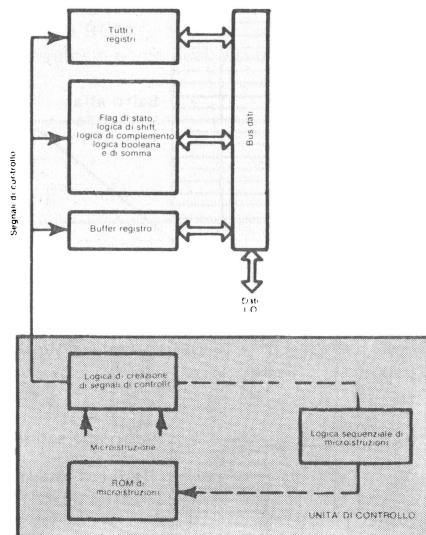
Non possiamo imporre limitazioni al set di istruzioni; se lo facciamo, i blocchi fondamentali della CPU non saranno universali. Perciò, incominciamo col separare la logica dell'unità di controllo dal resto della CPU:

Ora, se tornate indietro a fare riferimento alla nostra discussione sulla microprogrammazione, vedrete che un'unità di controllo, in realtà, consiste di un microprogramma memorizzato in memoria di sola lettura:

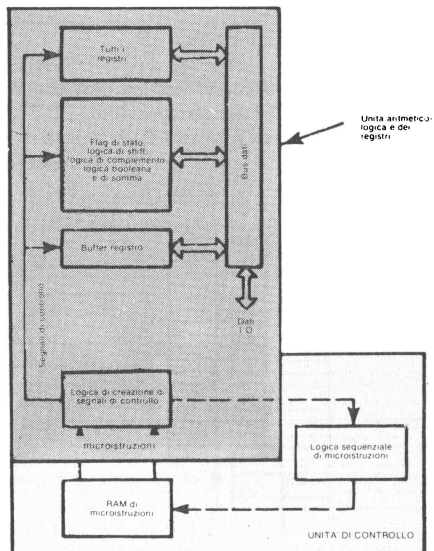
L'area scura chiamata "microprogramma" contiene sequenze di microistruzioni esattamente come le sequenze illustrate nella Tabella 4-5 a 4-8. CU DATA rappresenta un piccolo spazio di lavoro della memoria di lettura/scrittura, necessario all'unità di controllo.

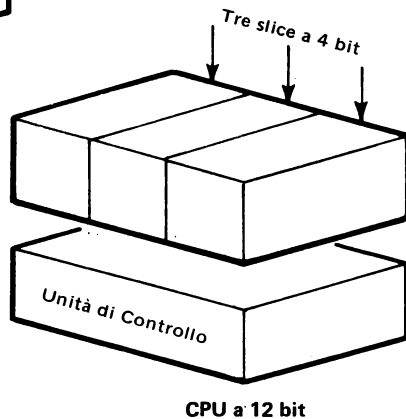
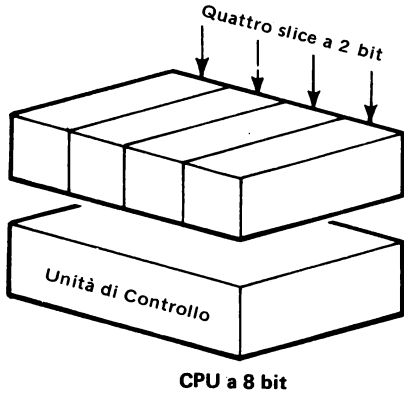
Ciò che abbiamo ignorato finora è la logica che vi permetterà di scegliere il modo di eseguire il microprogramma ROM — concatenando brevi sequenze di microistruzioni nel microprogramma di risposta a una qualsiasi macroistruzione:

Questa figura mostra arbitrariamente quattro sequenze separate di microistruzioni (scure), che devono essere eseguite allo scopo di abilitare la sequenza logica di eventi richiesta da alcune macroistruzioni indefinite. La linea tratteggiata identifica l'ordine nel quale devono essere eseguite le sequenze di macroistruzioni.

LOGICA DEL SEQUENZIATORE DEL MICROPROGRAMMA

La nostra unità di controllo deve contenere la logica del sequenziatore del microprogramma che gli permetta di trovare il modo di eseguire il microprogramma ROM, come mostra la linea tratteggiata nella figura. Vediamo alcune delle funzioni che la nostra logica del sequenziatore deve essere in grado di eseguire:

1) Deve accedere ad una sequenza contigua di microistruzioni, iniziando con una ben definita microistruzione e continuando per un numero fissato di microistruzioni:

2) Deve essere in grado di saltare ad un'altra sequenza di microistruzioni contigue:

3) Deve essere in grado di saltare ad una sequenza di microistruzioni usata di frequente, come per esempio, un accesso in memoria, e poi di ritornare al punto da cui ha effettuato il salto:

4) Deve essere in grado di rieseguire continuamente una sola microistruzione, come una "operation", per un numero prefissato di volte:

L'unità di controllo, in realtà, diventerà un microprogramma ROM con la associata logica sequenziale di microistruzioni:

CHIP SLICE DEI REGISTRI E DELL'UNITA' ARITMETICO-LOGICA

In pratica, è più facile implementare la logica che crea i segnali di controllo come parte dell'unità aritmetico-logica e dei registri:

Inizieremo la nostra discussione sui prodotti chip slice con l'unità aritmetico-logica e dei registri, che divideremo in segmenti.

Nel suddividere questa logica, dobbiamo assolutamente mettere il minor numero di restrizioni possibile sul numero e sull'organizzazione dei registri. Inoltre, non possiamo limitare la lunghezza di parole della CPU; anche se parliamo di microcomputer a 8 bit, sarebbe imprevidente supporre che la lunghezza di parola di 8 bit, sarebbe immutabile. Nell'industria dei microcomputer, si misura "sempre" in mesi, non in anni. **Suddivideremo quindi la nostra unità aritmetico-logica e dei registri in parti verticali identiche, tali che possano essere unite insieme per formare una CPU** con una qualunque lunghezza di parola:

PARTI DELLA ALU

Ognuna di queste parti sarà una parte della ALU. Le parti di ALU a 2 e a 4 bit sono di uso comune. Finchè la vostra lunghezza di parola è un multiplo di 4, la parte a 4 bit è migliore, dato che richiede meno chip.

Se è necessario suddividere la logica combinata dei registri e della ALU, ogni parte deve essere in grado di interfacciare con un elemento simile su entrambi i lati, in aggiunta all'unità di controllo.

Qualunque organizzazione della ALU anche semplice, come quella illustrata in Figura 4-1, presenta parecchi problemi. Gli innumerevoli canali che convergono sul bus dati vengono sempre più complessi, dato che i registri, se devono essere universali, non

possono essere predefiniti o limitati nel numero, come già visto. Dovreste costruire delle microistruzioni non certo pratiche per identificare le innumerevoli combinazioni di canale dati validi. Perciò riorganizzeremo i nostri registri e la nostra ALU cercando di disporre i canali dati, senza perdere la flessibilità. Ricordate, un chip slice ben riuscito non influisce sull'architettura del prodotto finito.

Attualmente esiste un grande numero di microcomputer basati sui microprocessori presenti sul mercato, e tale numero aumenta costantemente; ma vi sono pochissimi prodotti chip slice.
Perciò riorganizzeremo la porzione di ALU e di registri della Figura 4-1 per conformarci, in linea generale, con l'organizzazione dei prodotti chip slice a 4 bit della serie 2900 e 6700 che sono descritti nel Volume II. La Figura 4-3 illustra questa riorganizzazione. I prodotti chip slice a 2 bit serie 300 sono concettualmente simili, e rappresentano i predecessori della serie di prodotti 2900 e 6700. I prodotti chip slice 10800 rappresentano la generazione futura — la logica evoluzione della serie 2900 e 6700.

La Figura 4-4 illustra il concetto di un "chip slice"; la figura mostra due parti a 4 bit che creano una ALU a 8 bit.
Ritorniamo ora sulla Figura 4-3.

BLOCCO REGISTRI

Bisogna specificare un certo numero prefissato di registri **all'interno del blocco registri. Vengono scelti 16 registri,** dato che un codice di selezione a 4 bit può indirizzare uno dei 16 registri.

Il blocco registri ha due porte di output, AA e BB, più una porta di input, ZZ. Avendo tre porte, il blocco registri ha bisogno di tre set di logica selettiva dei registri — uno per ogni porta. La logica selettiva identifica il registro che è effettivamente collegato ad ogni porta in qualunque momento. Lo possiamo fare con due set di logica selettiva dei registri combinando la logica selettiva dell'input ZZ con la logica selettiva della porta di output AA o BB. Sceglieremo arbitrariamente di combinare la logica selettiva ZZ e BB. Ciò significa che, in qualunque momento, un medesimo registro del blocco registri verrà effettivamente collegato sia alla porta ZZ che alla porta BB.

Perciò il chip slice DIP ha bisogno di quattro pin di selezione del registro della porta A e quattro pin di selezione del registro della porta B:

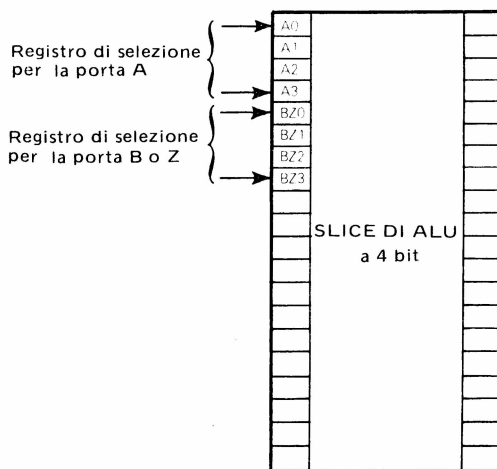
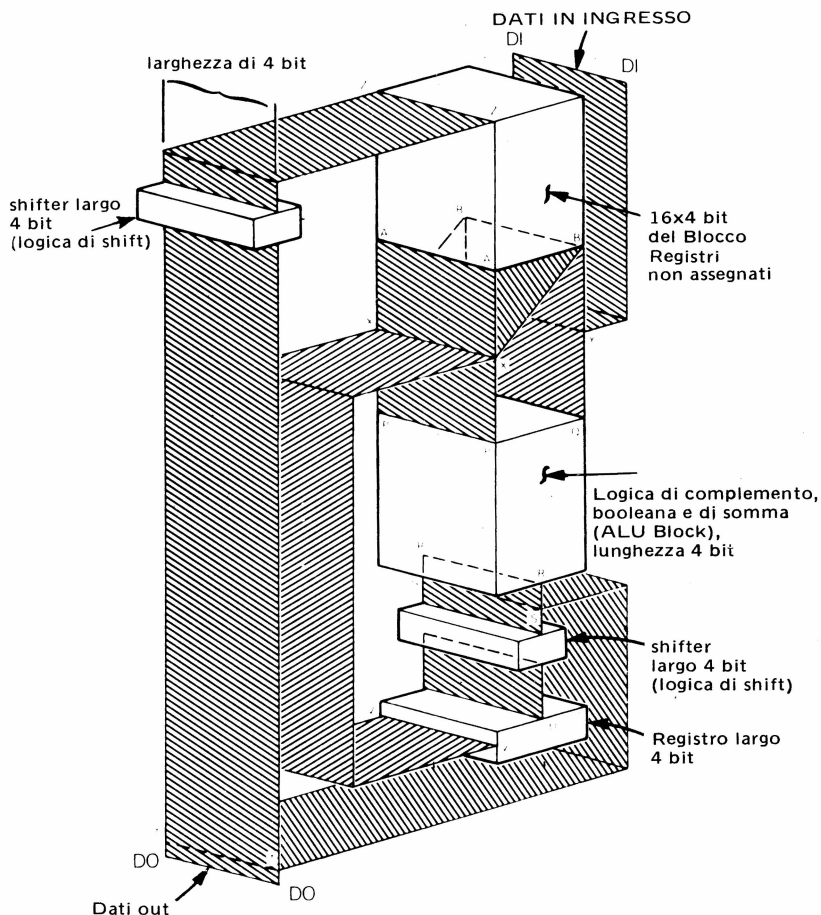



Figura 4-3. Unità logico-aritmetica e dei registri
Tratta della Figura 4-1 e riorganizzata per soddisfare le esigenze di un chip slice

CANALI DATI **Ora consideriamo i canali dati fra il blocco registri e il blocco ALU. Il blocco ALU richiede due porte d'ingresso, chiamate PP e QQ, dato che un certo numero di operazioni ALU richiedono due input per creare un output. RR indica l'output.**

La porta di input PP può ricevere l'output del blocco registri AA o BB, o può ricevere il contenuto del registro buffer, XX indica una giunzione a tre vie da cui deriva l'ingresso PP.

La porta d'ingresso QQ può ricevere o l'uscita BB del blocco registri, o i dati esterni. YY indica la giunzione di due tipi da cui deriva ingresso QQ.

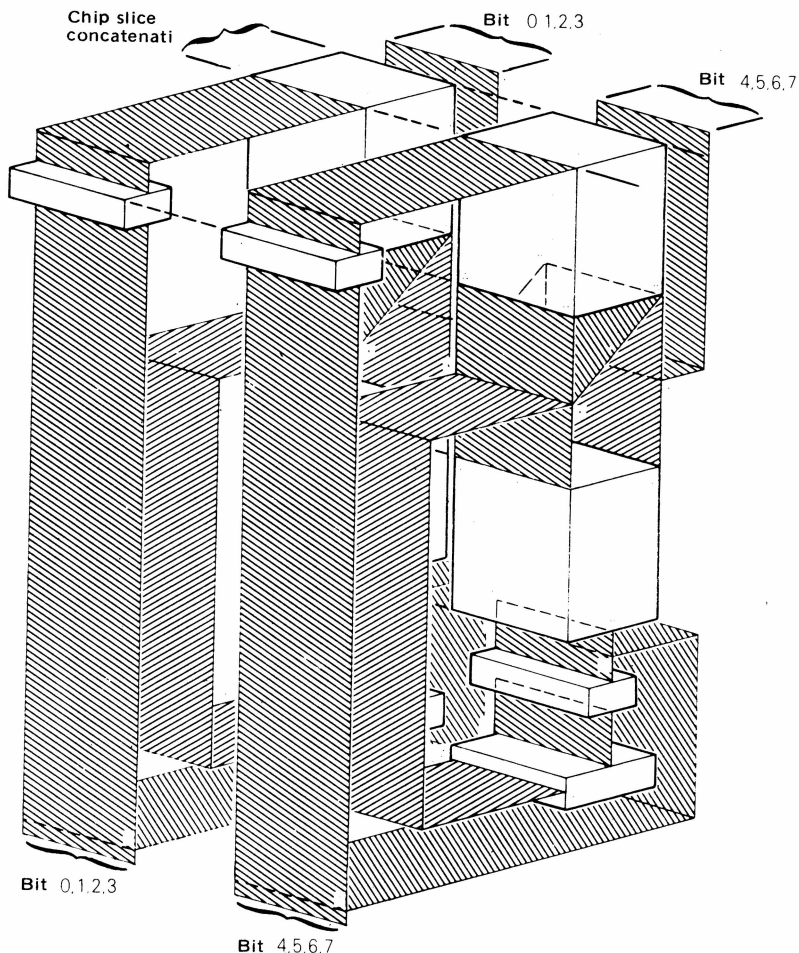



Figura 4-4. Due slice di ALU a 4 bit concatenati per generare un'ALU a 8 bit

INPUT IDENTIFICATO DELLA ALU

Esaminiamo i segnali esterni che saranno necessari per dare un supporto all'interfaccia fra il blocco registri e quello della ALU. Se supponiamo che un'opzione di addizione debba mettere in input 0 alle porte PP e QQ, **allora sono permesse le seguenti combinazioni di input:**

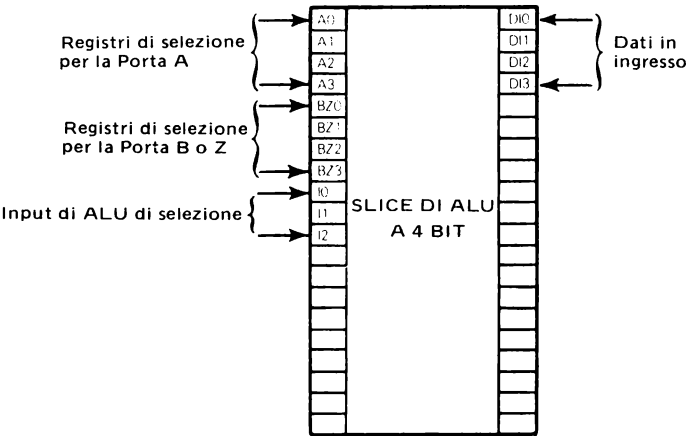
QQ:	0	0	0	0	BB	BB	BB	BB	DI	DI	DI	DI
PP:	0	AA	BB	VV	0	AA	BB	VV	0	AA	BB	VV

BB-0 è uguale a 0-BB, perciò ignoratelo.
Dato che AA e BB possono avere lo stesso valore, ignorate BB-BB che può essere fatto equivalere a BB-AA; ignorate DI--BB che può essere fatto equivalere a DI-AA.
0-0 è ignorato, dato che un'operazione ALU non avrà mai bisogno di due input 0.

Useremo tre pin di input per identificare le otto possibili combinazioni di input PP-QQ rimanenti. Questi pin di input diventeranno i tre bit di ordine inferiore di un codice microistruzioni a 9 bit e saranno interpretati come mostra la Tabella 4-9.

Tabella 4-9. Fonti di ALU così come definite dai tre bit di microistruzione meno significativi

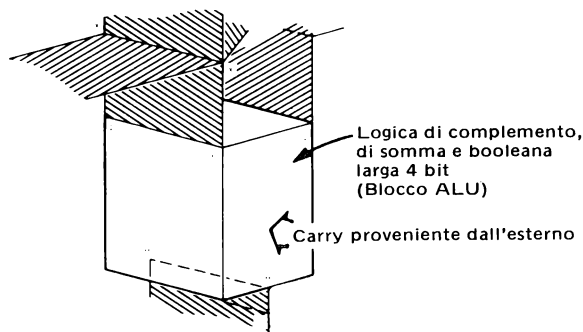
MICROISTRUZIONE			INPUT DI ALU	
I2	**I1**	**I0**	**QQ**	**PP**
0	0	0	BB	VV
0	0	1	BB	AA
0	1	0	00	VV
0	1	1	00	AA
1	0	0	00	BB
1	0	1	DD	BB
1	1	0	DD	VV
1	1	1	DD	00

L'interfaccia blocco registri-ALU richiede anche quattro dati nei pin per dare un supporto all'input dei dati in YY. Perciò la nostra DIP appare in questo modo:

UNITA' ARITMETICO-LOGICA DEL CHIP SLICE

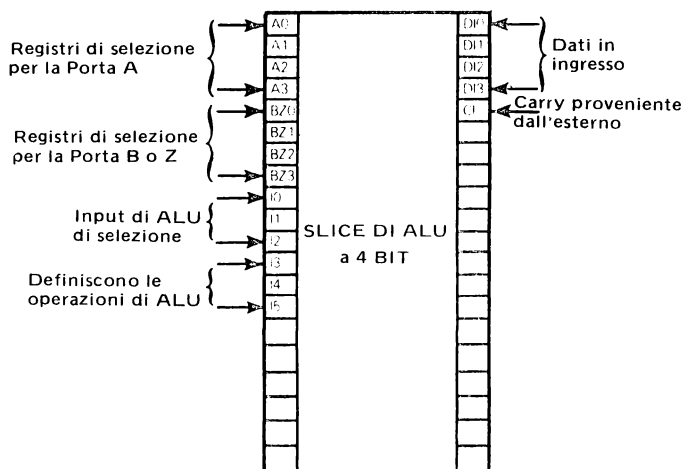
Ora spostiamoci nell'unità ariemtico-logica. Notate che lo shifter è stato rimosso, lasciandosi dietro il complementatore, l'addizione e la logica booleana. Rimuovendo la logica di shift, si possono spostare i dati all'interno di un breve canale di riciclaggio attraverso il registro buffer (RRSSUUVV-XXPP), oppure possiamo rispostare indietro verso il blocco registri (RRSSTTZZ), il risultato finale di un'operazione ALU.

Il registro buffer nella Figura 4-3 non serve allo stesso scopo per cui serviva nella Figura 4-1. Nella Figura 4-1 il registro buffer fornisce il secondo input ALU in qualunque momento un'operazione ALU richieda due input. Nella Figura 4-3 gli input ALU vengono dalle due porte di output del blocco registri PP e QQ. Nella Figura 4-3, il registro buffer è diventato una posizione che trattiene i risultati intermedi delle operazioni ALU.

IDENTIFICAZIONE DELLE OPERAZIONI ALU DI UN CHIP SLICE

Assegneremo i prossimi tre bit del codice microistruzioni (13, 14, 15) alla definizione delle operazioni ALU che devono essere eseguite. Vi sono cinque operazioni singole: ADD, COMPLEMENT, AND, OR, XOR. Potremmo aggiungere alla lista "incrementa" e "decrementa"; generiamo invece l'equivalente fornendo un Carry proveniente dall'esterno:

L'output RR risulta dagli input PP, QQ e dal Carry.

La combinazione delle due opzioni di input con il Carry e le cinque operazioni ALU, ci permette di generare i codici operativi della ALU illustrati nella Tabella 4-10.

La nostra parte di ALU a 4 bit, del DIP, avrà bisogno di altri tre input di microistruzioni, più un carry:

DESTINAZIONE DELLA ALU DEL CHIP SLICE

Resta da specificare solo la destinazione della ALU; useremo a questo scopo tre bit di microcodice.
Queste sono le tre possibili destinazioni per l'output del blocco ALU:

1) Il registro buffer, per mezzo di SS e UU.

2) Il blocco registri, per mezzo di SS, TT e ZZ.
3) I dati in output, per mezzo di SS e TT.

I dati che vanno verso il buffer o il blocco registri possono, a scelta, essere shiftati a sinistra o a destra. Si potrebbe selezionare una sbalorditiva quantità di opzioni di output dato che i dati possono essere messi in output su una qualunque o su tutte le tre destinazioni, effettuando gli spostamenti su due dei canali di destinazione. Finchè fate un larghissimo uso di un prodotto chip slice, non sarà chiaro quali delle opzioni dei canali di output siano utili.

Tabella 4-10. Operazioni di ALU specificate dai tre bit di microcodice mediani

MICROCODICE			**FUNZIONE**		
15	**14**	**13**	**Generale**	**Carry dall'esterno = 0**	**Carry dall'esterno = 1**
0	0	0	QQ + PP	QQ + PP PP se QQ è 0 QQ se PP è 0	QQ + PP + 1 Incrementano PP se QQ = 0 Incrementano DD se PP = 0
0	0	1	QQ + ($\overline{PP}$) ($\overline{PP}$ è il complemento a uno di PP)	QQ - PP + 1 PP complemento a a uno se QQ = 0	PP - QQ PP complemento a a due se QQ = 0
0	1	0	QQ OR PP	Il carry proveniente dall'esterno non riveste nessuna funzione nelle operazioni di logica booleana	
0	1	1	QQ AND PP		
1	0	0	QQ XOR PP		
1	0	1	non assegnati al momento		
1	1	0			
1	1	1			

Tabella 4-11. Destinazioni di ALU specificate dagli ultimi bit di microcodice

18 17 16	**REGISTRO BUFFER**		**BLOCCO REGISTRI**		**Dati in uscita**
	Shift		**Shift**		
0 0 0	No	Si		No	Si
0 0 1	Sinistra	Si		No	Si
0 1 0	Destra	Si		No	Si
0 1 1		No		No	Si
1 0 0		No	No	Si	Si
1 0 1		No	Sinistra	Si	Si
1 1 0		No	Destra	Si	Si
1 1 1	No	Si	No	Si	Si

Ricordate che in realtà abbiamo due tipi di output ALU — i dati temporanei che si dirigono verso il registro buffer e le risposte permanenti che tornano al blocco registri. Basandosi su questo concetto la Tabella 4-11 illustra un modo in cui si possono specificare le destinazioni.

STATO DEL CHIP SLICE

L'unico argomento rimasto da discutere è lo stato. I flag di stato Zero, Overflow e Segno sono facili da generare, perciò vediamo prima questi tre.

STATO DI SEGNO

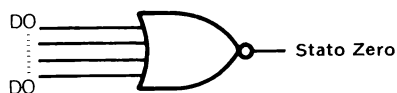
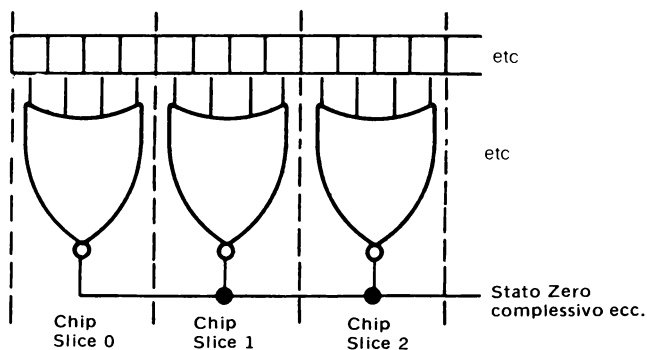
Ogni chip slice sarà costruito supponendo che possa essere la parte di ordine superiore della ALU. Ogni parte avrà quindi una logica che suppone che le linee di dati rappresentino i quattro bit di ordine superiore della eventuale parola della ALU. La linea di ordine superiore rappresenterà perciò il bit del segno:

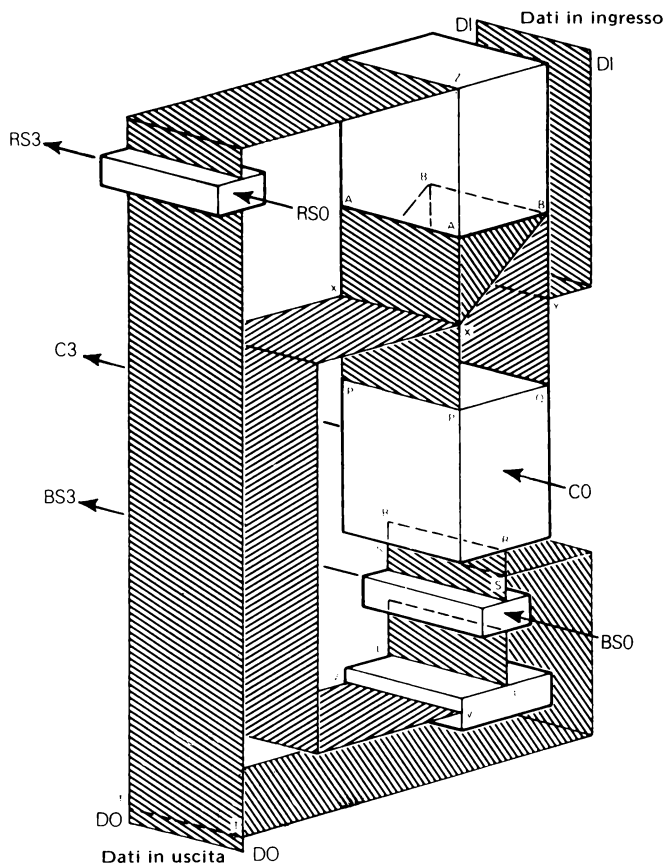
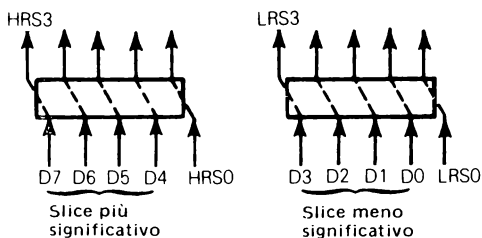
Possiamo generare il bit del segno direttamente dalla linea dati di ordine superiore di ogni singolo chip slice. Verrà usato solo il bit del segno del chip slice di ordine superiore; gli altri bit del segno del chip slice saranno ignorati.

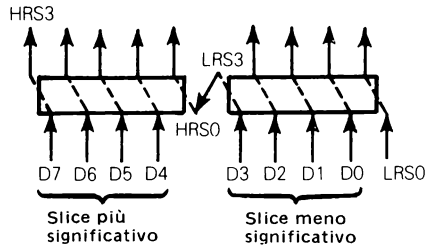
STATO DI OVERFLOW

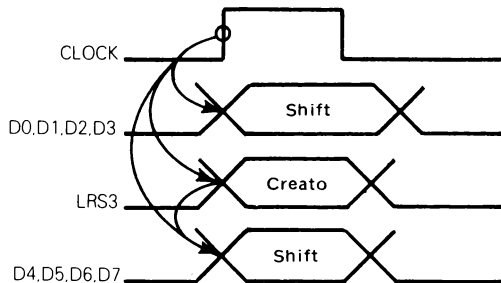
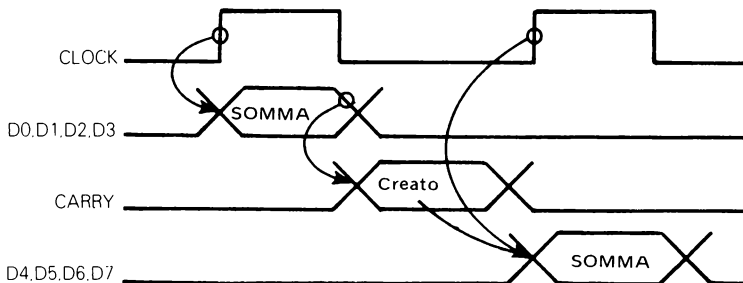
Lo stato di Overflow può essere generato dalle due linee di ordine superiore di ogni dato del chip slice, come lo stato del segno veniva generato dalla linea di ordine superiore.

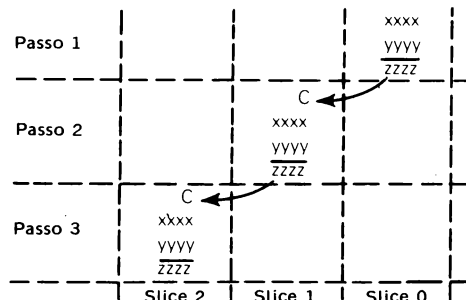
Come descritto nel Capitolo 2, lo stato Overflow rappresenta l'OR esclusivo dei Carry del penultimo e dell'ultimo bit di una parola dati. QUINDI LA LOGICA DI OVERFLOW PUO' ESSERE GENERATA SOLO ALL'INTERNO della ALU.

STATO ZERO **Anche generare lo stato Zero è un processo diretto. Per ogni chip slice metteremo in output NOT OR delle quattro linee di dati.**

Legando gli stati Zero di tutti i chip slice all'interno della CPU insieme fra di loro, potete creare uno stato Zero complessivo:

Gli stati Carry e Shift non sono così immediati. Prima di tutto ci occorrono tre set di stati.

Dati in ingresso
DI
DI
RS3
RS0
C3
BS3
C0
BS0
DO
Dati in uscita
DO

Ma gli stati illustrati nella figura non funzioneranno sempre in maniera efficiente, dato che i chip slice devono lavorare in parallelo.
Consideriamo un semplice shift di 8 bit, creato usando due chip slice:

Se lo shift illustrato nella figura deve avvenire con un solo passo parallelo, la parte del chip slice di ordine inferiore persa (LRS3) deve diventare la parte del chip slice di ordine superiore acquisita (HRSO).

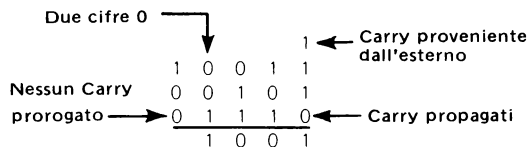
Se LRS3 e HRSO sono entrambi collegati ai pin DIP, tutto quello che dobbiamo fare è collegare questi due pin e si ha quindi uno shift a 8 bit:

Ma quando si arriva all'addizione binaria, il problema è meno semplice. Quando si fa uno shift, LRS3 viene creato mentre lo spostamento è in corso. Quando si fa un'addizione, il Carry viene generato alla fine dell'addizione stessa.

Ecco un'illustrazione semplificata di questo problema di timing. Per uno shift, non abbiamo problemi:

Per l'addizione binaria, abbiamo un problema:

STATO CARRY

Potremmo eseguire l'addizione binaria con incrementi di 4 bit, iniziando con i quattro bit meno significativi; e in questo caso, il riporto potrebbe semplicemente essere riportato da ogni parte a 4 bit alla successiva:

CONTROLLO ANTICIPATO DEL CARRY

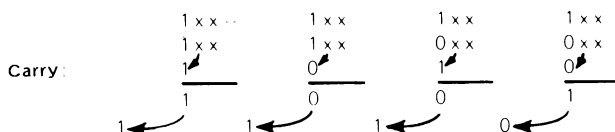
Ma il fatto di suddividere in parti l'addizione binaria fa fallire lo scopo di usare prodotti chip slice — cioè di guadagnare velocità nell'esecuzione delle istruzioni. Dobbiamo perciò aggiungere della logica che permetta alla ALU di prevedere se un'addizione binaria sta per creare un Carry, o per propagare un Carry in entrata.
Le regole per la creazione e la propagazione del riporto sono abbastanza semplici.

PROPAGAZIONE DEL CARRY

Consideriamo prima la propagazione del Carry. Se vi è un Carry di un'addizione binaria, vi sarà propagazione finchè non si dovranno sommare due cifre 0 — rompendo così la catena di propagazione:

Se P_i e Q_i rappresentano cifre binarie che entrano nella ALU attraverso le porte PP e QQ rispettivamente, concludiamo che un riporto si propagherà se:

$$(P_0 \text{ OR } Q_0) \text{ AND } (P_1 \text{ OR } Q_1) \text{ AND } (P_2 \text{ OR } Q_2) \text{ AND } (P_3 \text{ OR } Q_3) = 1$$

GENERATORE DEL CARRY

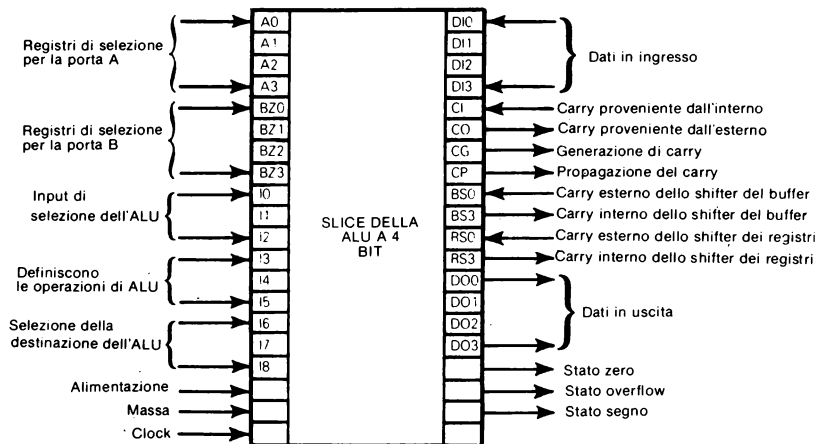
Per determinare se verrà generato un nuovo Carry, dobbiamo partire dall'estremità di ordine superiore dell'unità a 4 bit e lavorare andando all'indietro fino alla estremità di ordine inferiore. Entrambe le cifre di ordine superiore devono essere uno o una cifra di ordine superiore deve essere 1 con un Carry propagato dalle penultime cifre:

Se C_i rappresenta il Carry della posizione di bit 1, allora se $C_i = 1$, si genera un Carry:

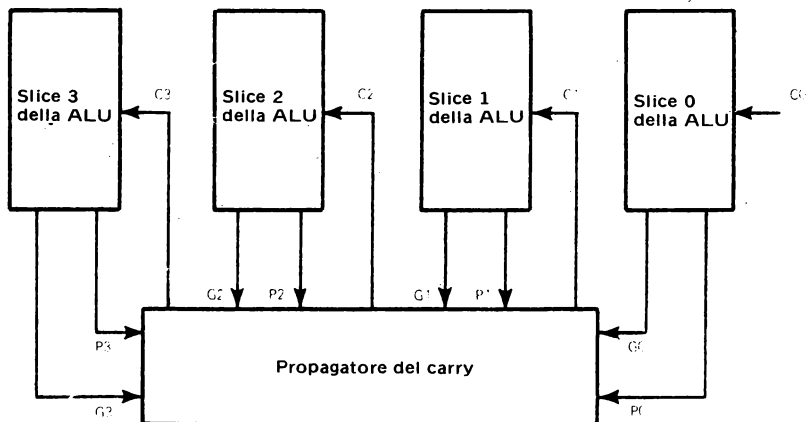
$$(P_3 \text{ AND } Q_3) \text{ OR } (C_2 \text{ AND } (P_2 \text{ OR } Q_2)) = 1$$

Per C_2 a = 1, si applica la stessa relazione, con le posizioni dei bit spostate verso il basso di 1

$$(P_2 \text{ AND } Q_2) \text{ OR } (C_1 \text{ AND } (P_1 \text{ OR } Q_1)) = 1$$

In questo modo la ALU può essere fornita di una logica che prevede la generazione del Carry.

Per finire, questo è il modo in cui i pin devono essere assegnati:

DISPOSITIVI DI GENERAZIONE DEL CARRY

La logica che genera il Carry sarà solitamente fornita su di un dispositivo separato costruito per generare il Carry. Questo dispositivo riceve i segnali di propagazione del Carry (P) e di generazione del Carry (G), nell'appropriata sequenza, dalle parti della ALU a 4 bit; esso genera il Carry (C) corretto e lo rimanda ad ogni chip slice:

L'UNITA' DI CONTROLLO DEL CHIP SLICE

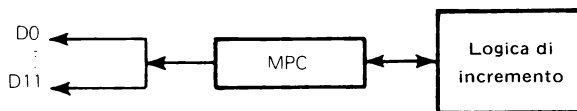
Le parti della ALU, come le abbiamo descritte, sono guidate da microistruzione di 9 bit, insieme con gli input dei dati binari e con i vari segnali di controllo e di stato.

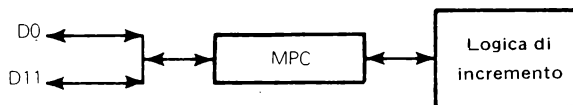
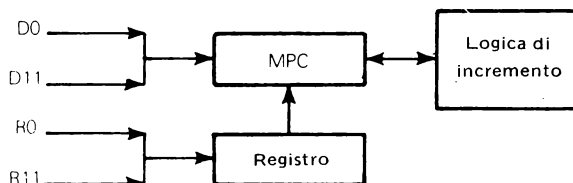
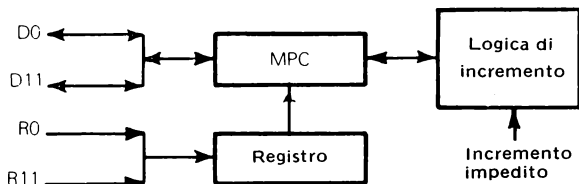
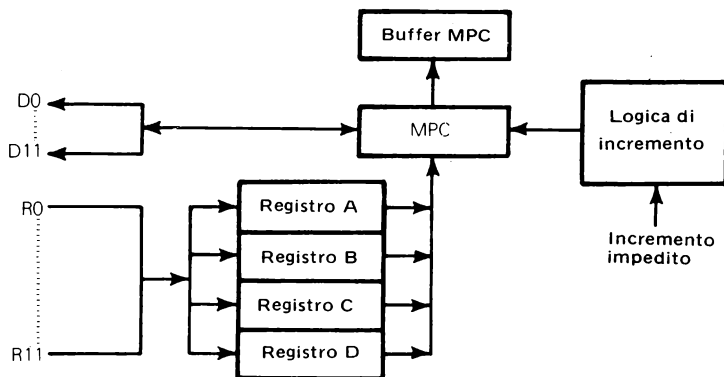
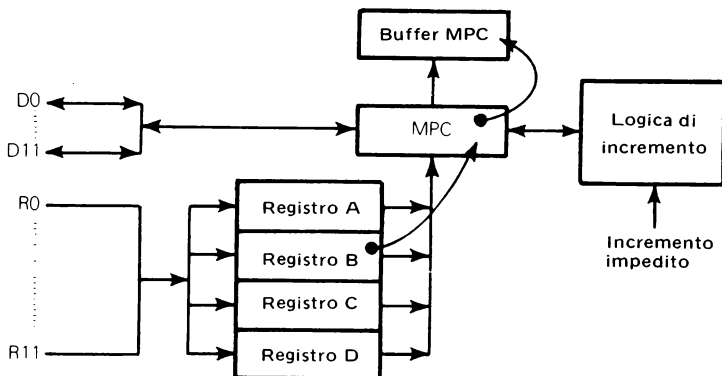
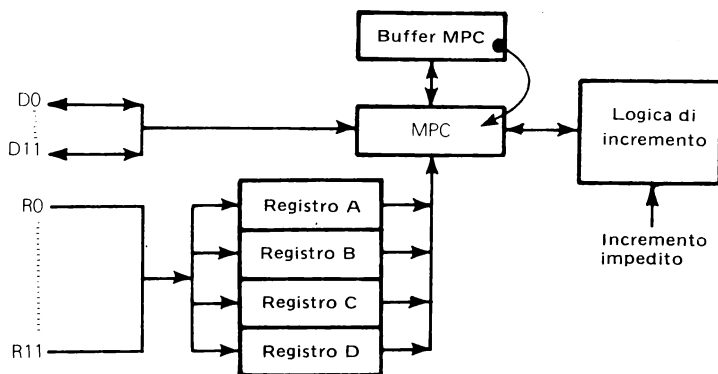
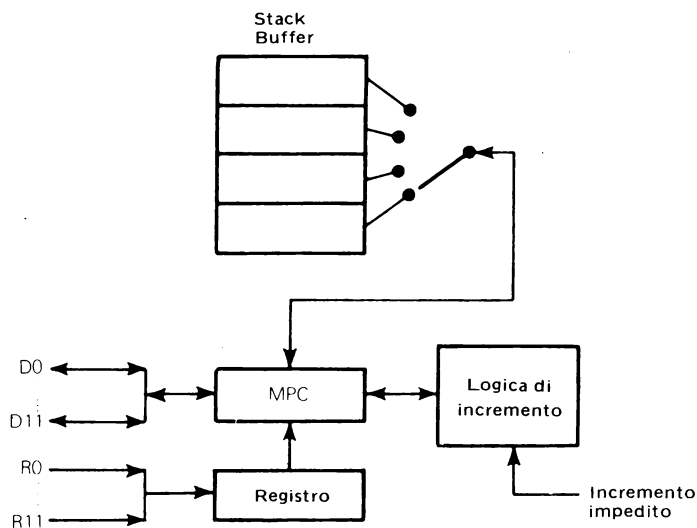
L'unità di controllo deve fornire il codice di microistruzione a 9 bit; potrebbe fornire anche i segnali di input di stato/controllo solitamente non lo fa, per motivi di cui parleremo presto.

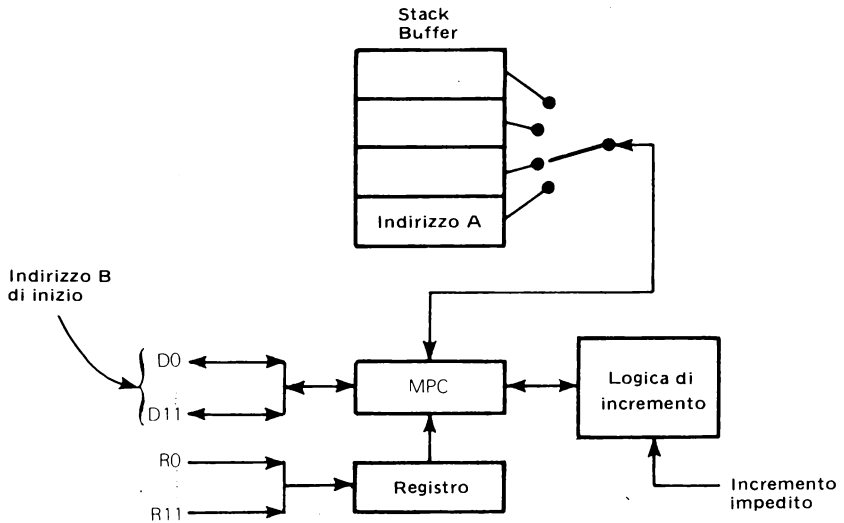
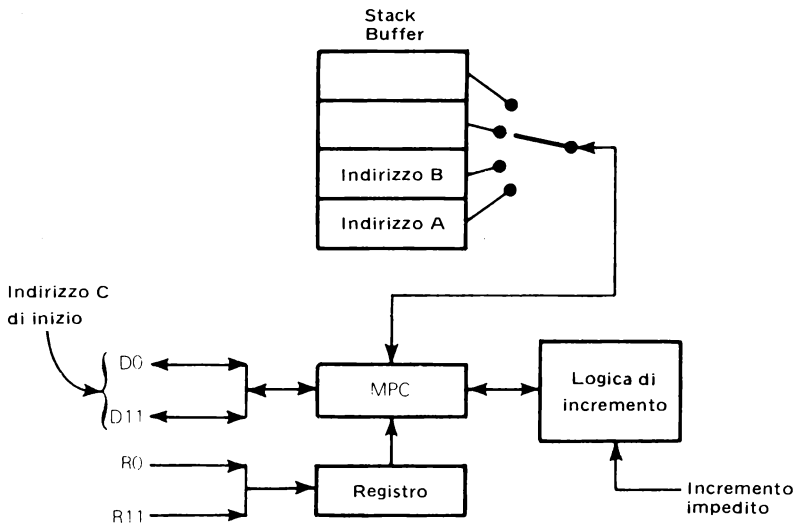
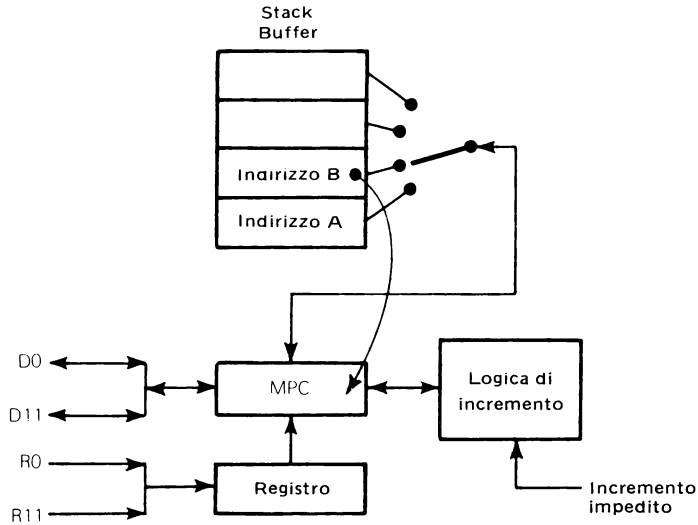
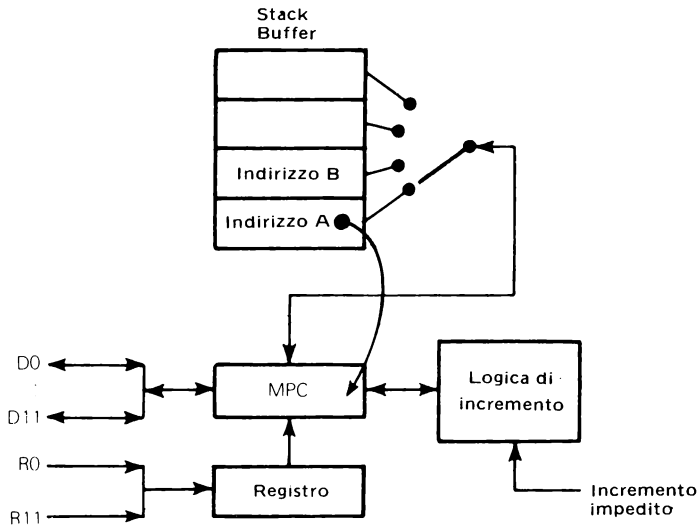
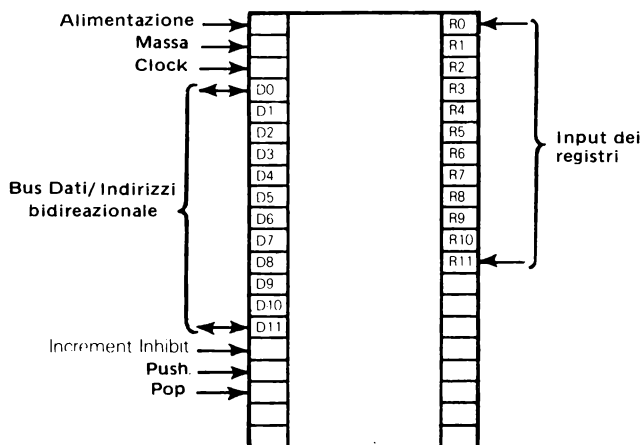
Stiamo per memorizzare il codice di microistruzione in una memoria di sola lettura molto veloce e per creare una logica d'indirizzamento che accede alle microistruzioni nell'appropriata sequenza. **L'unità di controllo consiste allora di microistruzioni ROM e della loro logica di indirizzamento, come abbiamo già detto in questo Capitolo.**

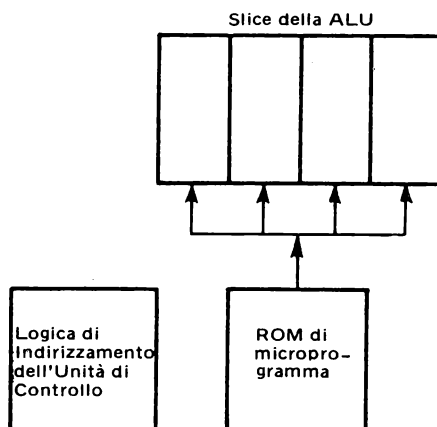
Possiamo osservare parecchie cose nelle caratteristiche della logica di indirizzamento dell'unità di controllo tornando a guardare le sequenze di microistruzioni del microprocessore nelle Tabelle da 4-5 a 4-8.

CONTATORE DI MICROPROGRAMMA

In circostanze normali si accede in modo sequenziale ai codici delle microistruzioni. Perciò, **la logica di indirizzamento dell'unità di controllo deve avere un contatore di microprogramma (MPC),** cioè l'equivalente di un Program Counter che può essere incrementato dopo ogni accesso al microprogramma per fare riferimento alla macroistruzione sequenziale successiva:

Supponiamo arbitrariamente che la logica d'indirizzamento dell'unità di controllo abbia una larghezza di 12 bit comprendendo un massimo di 4096_{10} microistruzioni della ROM. Ogni sequenza di microistruzioni incomincerà ad un certo indirizzo; percio **la logica d'indirizzamento dell'unità di controllo deve essere in grado di inizializzare il Microprogram Counter.** Consideriamo due possibilità:

1) Ogni codice oggetto di macroistruzione sta per essere implementato da una sequenza di microistruzioni per mezzo del suo indirizzo iniziale che deve quindi essere caricato nel Microprogram Counter. **Forniremo quindi la possibilità di accesso diretto al Microprogram Counter.**

2) Sarebbe auspicabile conoscere l'origine di alcuni programmi di uso generale per trattare circostanze speciali o condizioni di allarme che possono non aver niente a che fare con l'esecuzione di una particolare istruzione. **Forniremo quindi un registro in cui possano essere memorizzati alcuni du questi indirizzi permanenti:**

Ricordate che la logica di indirizzamento dell'unità di controllo deve essere in grado di rieseguire un'istruzione per un certo numero di volte. Nel nostro, è stata rieseguita un'istruzione di "no operation" semplicemente mantenendo l'unità di controllo sincronizzata con il Timing esterno. **Aggiungeremo quindi un controllo di inibizione dell'incremento al Microprogram Counter:**

Infine, **ricordate che vi sono sequenze di istruzioni di uso comune che eseguono operazioni quali una lettura o una scrittura in memoria.**
Possiamo trattare la cosa in uno dei due modi seguenti. Prima, consideriamo di avere un numero di registri d'indirizzamento più un buffer Microprogram Counter.

Come illustra la figura, l'indirizzo della prima microistruzione di quattro sequenze di microistruzioni usate spesso, può essere memorizzato nei registri A, B, C e D. La logica d'indirizzamento dell'unità di controllo può prima salvare il contenuto del MPC nel buffer, poi caricare il contenuto di un registro:

L'ultima microistruzione della sequenza di uso comune fa sì che il contenuto del buffer venga rimandato al Microprogram Counter.

Il secondo approccio richiede della logica esterna per fornire l'indirizzo di inizio di ogni sequenza di microistruzioni usata di frequente. In questo caso, **uno stack di registri buffer** supporterà il Microprogram Counter.

Lo stack buffer permette ad una sequenza di microistruzioni frequentemente usata di accedere ad un'altra sequenza di microistruzioni usata di frequente. Questo è detto annidamento delle sequenze. Lo stack è una normale caratteristica dei microcomputer ed è descritto nel Capitolo 6.
Qui di seguito è illustrato il modo in cui lo stack buffer lavorerà per far accedere la sequenza di microistruzioni A alla sequenza di microistruzioni B, che, a sua volta, accede alla sequenza di microistruzioni C.

1) La sequenza di microistruzioni A raggiunge il punto in cui deve accedere alla sequenza di microistruzioni B. L'indirizzo corrente della sequenza A viene salvato nello stack e viene preso in input l'indirizzo della sequenza B:

2) La sequenza di microistruzioni B raggiunge il punto in cui deve accedere alla sequenza di microistruzioni C. Si ripete il passo 1:

3) La sequenza di microistruzioni C completa l'esecuzione e poi l'indirizzo conservato di B viene rimandato a MPC:

4) La sequenza di microistruzioni B completa a sua volta l'esecuzione, e quindi, l'indirizzo conservato di A viene rimandato a MPC:

Supponendo che la logica d'indirizzamento della nostra unità di controllo abbia uno stack buffer, **saranno necessari due segnali di controllo addizionali:** uno metterà i contenuti di MPC nello stack, come è illustrato nei passi 1 e 2; l'altro preleverà l'indirizzo della parte superiore dello stack in MPC, come illustrano i passi 3 e 4. **Ora l'assegnazione dei pin della logica d'indirizzamento DIP della nostra unità di controllo avverrà nel modo seguente:**

L'ACCOPPIAMENTO DELL'UNITA' ARITMETICO-LOGICA CON L'UNITA' DI CONTROLLO

Concettualmente, costruiremo **un'unità centrale combinando parti di ALU con la logica d'indirizzamento dell'unità di controllo ed un microprogramma in memoria di sola lettura,** come segue:

In pratica, sarà necessaria una notevole quantità di logica esterna prima che la semplice configurazione illustrata nella figura possa lavorare come un'unità centrale. Per esempio, non abbiamo analizzato ancora il problema di ricevere o trasmettere segnali di controllo. Che dire dei segnali di controllo di READ e WRITE dei microprocessori?

Sarebbe stato possibile aggiungere della logica all'unità di controllo che automaticamente legge e crea i segnali di controllo tipo CPU. Comunque, ciò presuppone che i prodotti chip slice vengano usati solo come blocchi fondamentali della CPU. Tale presupposto è ingiustificato.

Descrivendo i prodotti chip slice nel Capitolo 4, un capitolo dedicato alle unità centrali, presentiamo i prodotti chip slice come blocchi fondamentali della CPU, il che li rende concettualmente facili da capire solo grazie alla sequenza secondo la quale le informazioni vengono presentate in questo libro.

Escludere la logica di elaborazione dei segnali di controllo dall'unità di controllo dei chip slice significa un enorme lavoro extra e logica in più che deve affiancare il set di chip slice e unità di controllo. Ma, allo stesso tempo, non vengono imposte limitazioni sul modo in cui vengono usati questi prodotti.

In riferimento all'argomento che stiamo trattando, quindi, dobbiamo concludere senza dimostrare l'equivalente di un prelevamento d'istruzione o di una tipica esecuzione d'istruzione, perchè il tipo di informazioni che dovrebbe essere studiato prima che si possa trattare in modo adeguato la logica esterna necessaria, va al di là degli scopi di questo libro.

Capitolo 5
LOGICA ADDIZIONALE DELLA CPU

In questo capitolo identificheremo la logica addizionale che deve affiancare una CPU allo scopo di dare vita ad un sistema microcomputer che sia abbastanza esteso da essere utile.

Dobbiamo identificare separatamente i componenti logici di un sistema e la funzione (es. CPU, memoria RAM, memoria ROM, ecc.), ma non vi è una correlazione fondamentale necessaria fra i componenti logici e i singoli chip. Come vedrete nel Volume II, ci sono molte differenze nella logica che ogni costruzione decide di mettere in un singolo chip.

MEMORIA DI PROGRAMMA E MEMORIA DATI

La memoria esterna è la prima e più ovvia aggiunta necessaria per dare un supporto alla CPU descritta nel capitolo 4.

MEMORIA DI SOLA LETTURA (ROM)

Interfacciare una ROM con una CPU è molto semplice.

Come descritto nel Capitolo 3, intere parole di memoria vengono implementate su di un solo chip ROM. Al contrario, la memoria di lettura/scrittura (RAM) può richiedere un chip diverso per ogni bit della parola di memoria.

I segnali richiesti da un dispositivo ROM sono assolutamente elementari. Come logicamente ci si aspetterebbe, un dispositivo ROM richiederà in input i seguenti segnali:

1) L'indirizzo della parola di memoria cui bisogna accedere.
2) Un segnale di controllo di lettura che dice al dispositivo ROM quando rimandare il contenuto della parola di memoria indirizzata.
3) Un segnale di clock in modo che ROM e CPU siano sincronizzate.
4) Alimentazione e massa.

Gli unici segnali di output che la ROM deve avere sono otto linee-dati (per una parola di 8 bit), per mezzo delle quali il contenuto della parola di memoria indirizzata può essere trasferita alla CPU. La Figura 5-1 illustra un'ipotetica ROM. Questa ROM è collegata alla CPU come illustra la Figura 5-2.

Prima di descrivere dettagliatamente un accesso alla ROM, consideriamo alcune delle caratteristiche non ovvie di un sistema microcomputer.

BUS DATI ESTERNO

Se un sistema microcomputer consistesse soltanto della CPU e di una ROM, i pin dei due dispositivi potrebbero essere collegati direttamente tra loro. Dato che deve essere possibile che in un sistema microcomputer stiano più di due dispositivi, i collegamenti dei segnali vengono fatti attraverso un bus dati esterno, che può essere paragonato ad un comune percorso che collegano i chip del set del microcomputer.

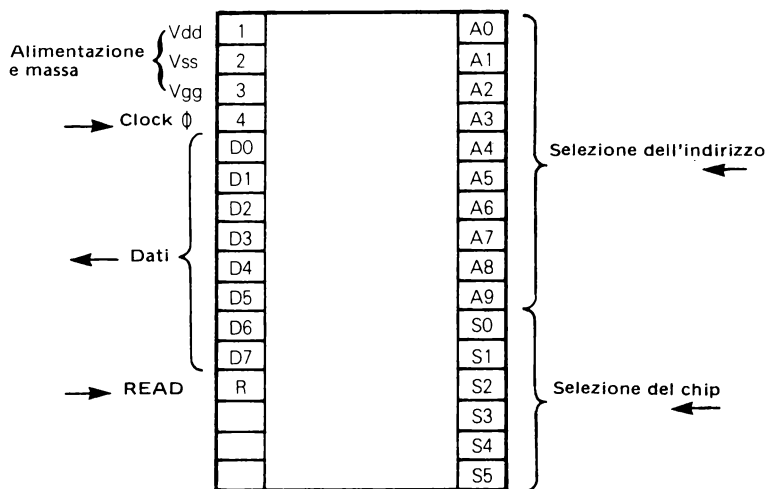



Figura 5-1. Segnali e Pin della memoria a sola lettura ROM

SELEZIONE DEI DISPOSITIVI ROM

Notate che i 16 segnali d'indirizzamento della CPU diventano dieci segnali di indirizzo di parola e sei segnali di selezione ROM. Questa distribuzione delle linee d'indirizzamento implica che la ROM abbia 1024 (2^{10}) parole di memoria, ed un codice di selezione di 6 bit. Inviando le sei linee di ordine superiore dell'indirizzo coincidenti con il codice dispositivo a 6 bit della ROM, questo decodificherà le dieci linee d'indirizzamento di ordine inferiore come rappresentanti una delle sue 1024 parole di memoria. Poi, quando il segnale di controllo di READ diventerà 1 (vero), la logica della ROM metterà il contenuto della parola di memoria indirizzata ai pin dati da D0 a D7. Se le sei linee d'indirizzamento di ordine superiore non coincidono con il codice di selezione della ROM, la ROM non esegue alcuna operazione.

Spesso il chip ROM non avrà in sè la logica di selezione del chip; allora vi sarà un solo segnale di selezione del chip, che deve essere generato dalla logica esterna. E' molto comune, per una ROM, avere anche due input di selezione. Perchè la ROM sia selezionata, un input deve essere basso, quando l'altro è simultaneamente alto.

Se il chip ROM ha una sua logica di selezione, il codice di selezione è una caratteristica permanente del progetto del chip della ROM stessa. Se, ad esempio, il chip ROM ha codice di selezione 001100, questo codice di selezione sarà stato impresso nella ROM quando essa è stata costruita, con il risultato che il chip ROM risponderà solo agli indirizzi di memoria da 3000_{16} a $33FF_{16}$:

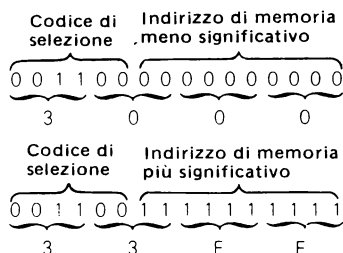
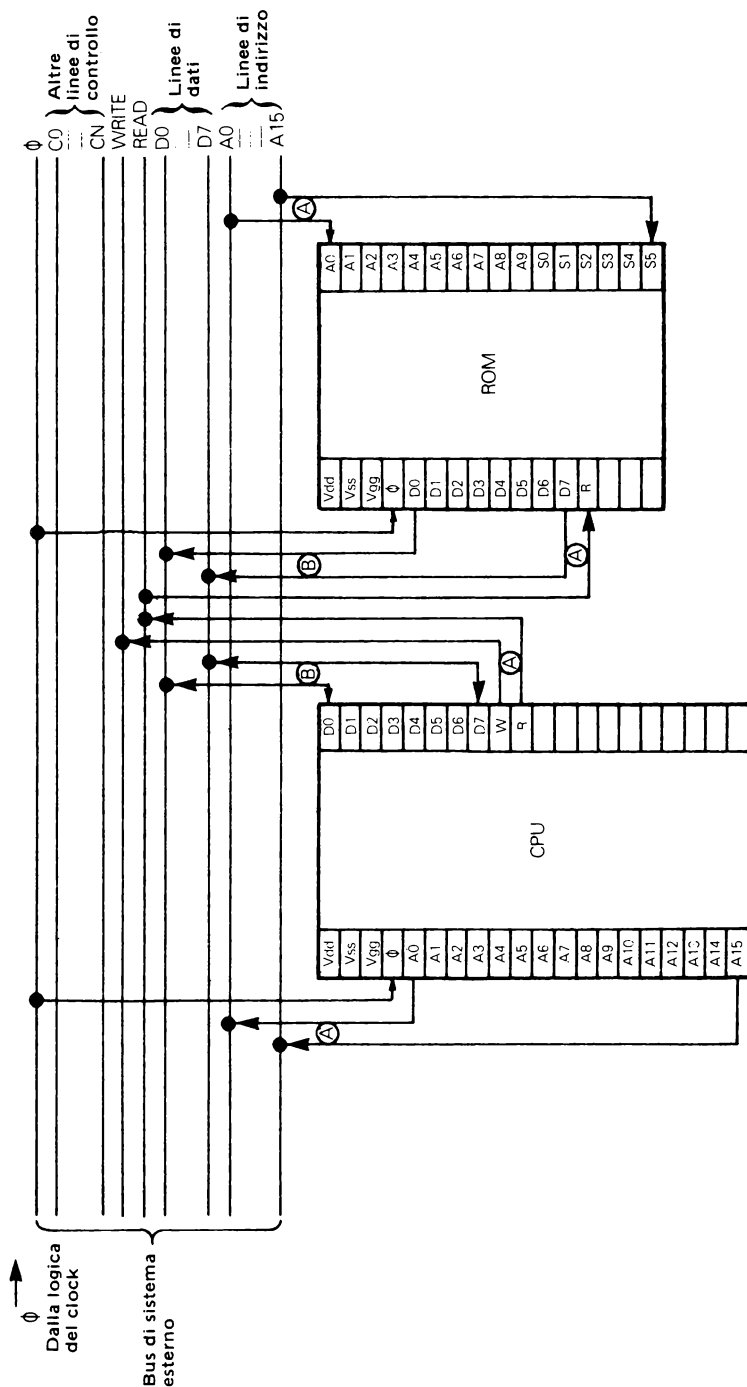
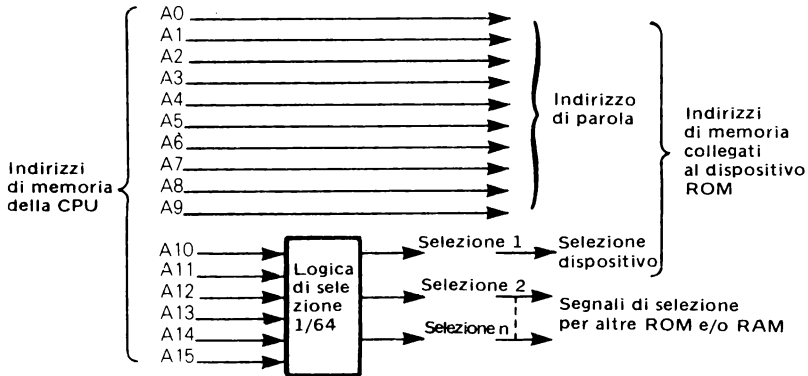
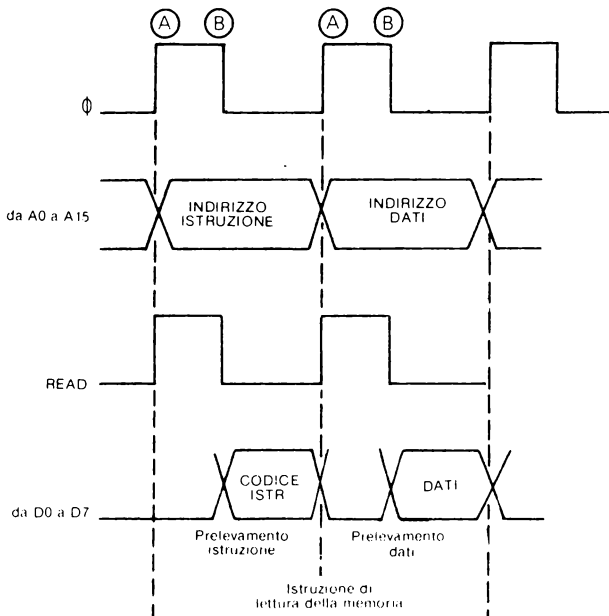



Figura 5-2. ROM e CPU collegate tramite Bus Dati esterno

In altre parole, una ROM che ha un certo codice di selezione deve essere considerata in modo diverso da una ROM identica che ha un diverso codice di selezione.

Se la logica di selezione dispositivo è esterna alla ROM, il codice di selezione non è una caratteristica permanente della ROM; modificando semplicemente la logica di selezione esterna si cambierà la serie di indirizzi entro la quale la ROM risponde agli accessi in memoria. La logica di selezione dispositivo esterna può essere illustrata in questo modo:

La logica interna ad una ROM non risulterà mai il vostro interesse, come utente di un microcomputer. Il modo in cui la ROM si seleziona o disseleziona da sola, il, modo in cui risponde ai segnali di controllo di lettura, il modo in cui estrae i dati necessari e li pone ai pin D0 al pin D7, sono del tutto irrilevanti dato che non potete farci nulla.

Consideriamo un'istruzione di lettura in memoria; vedete qui riprodotto il Timing per questa istruzione, come appariva nel Capitolo 4, ma con i simboli chiave Ⓐ e Ⓑ per collegarlo alla Figura 5-2.

Ricordate che, quando si parla di logica esterna alla CPU, **non esiste differenza fra un'operazione di prelevamento istruzione e un'operazione di prelevamento dati.**

Ogni operazione inizia col fronte di salita Ⓐ del clock Φ; nel frattempo la CPU mette in output un indirizzo sulle linee d'indirizzamento, e allo stesso tempo posiziona READ. La ROM riceve questi segnali attraverso il bus dati esterno. Se le sei linee d'indirizzamento di ordine superiore (A10-A15) coincidono con il codice di selezione della ROM (S0-S5) allora la logica della ROM preleva il contenuto della parola di memoria indirizzata da A0-A9, e mette i dati a D0-D7.

Quando Φ e READ si abbassano Ⓑ, la logica della ROM deve aver già posto i dati richiesti su D0-D7, dove devono stare finchè Φ torna nuovamente alto.

MEMORIA DI LETTURA/SCRITTURA (RAM)

La logica di interfaccia della RAM è più complessa della logica di interfaccia della ROM. La logica della RAM deve essere in grado di prelevare i dati dalle linee dati del bus del sistema esterno e di mettere questi dati in una parola di memoria indirizzata; inoltre la logica della RAM deve essere in grado di estrarre i dati da una parola di memoria indirizzata e di metterli sulle linee dati del bus del sistema esterno. Inoltre, la maggior parte delle RAM si implementano usando un numero di chip RAM con ogni chip che fornisce uno o più di ogni parola dati.

L'uso di un certo numero di chip per formare una sola parola dati è un concetto abbastanza semplice; significa che ogni chip avrà solo un pin dati, e questo pin sarà collegato ad una delle otto linee dati, D0-D7.

Come per la ROM, la logica di interfaccia della RAM suddividerà le linee d'indirizzamento A0-A15 in un codice di selezione dispositivo e in un indirizzo di memoria. Comunque, vi possono essere otto chip RAM (per una parola a 8 bit), ognuno dei

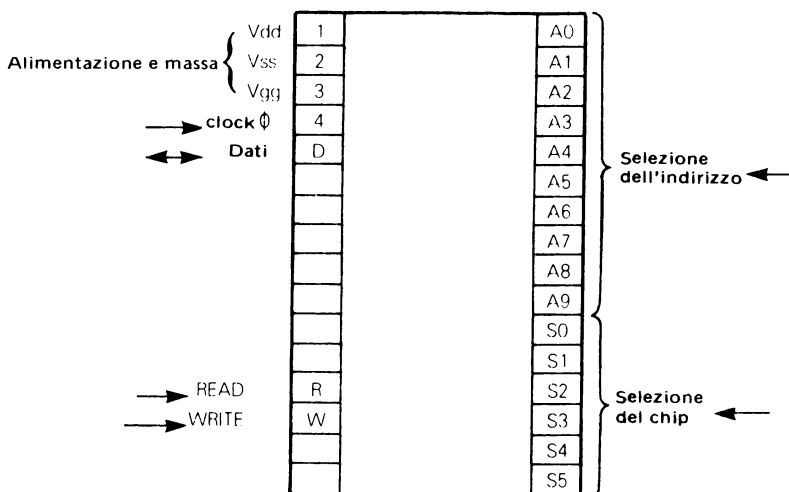



Figura 5-3. Piedini e segnali del chip della memoria a lettura/scrittura

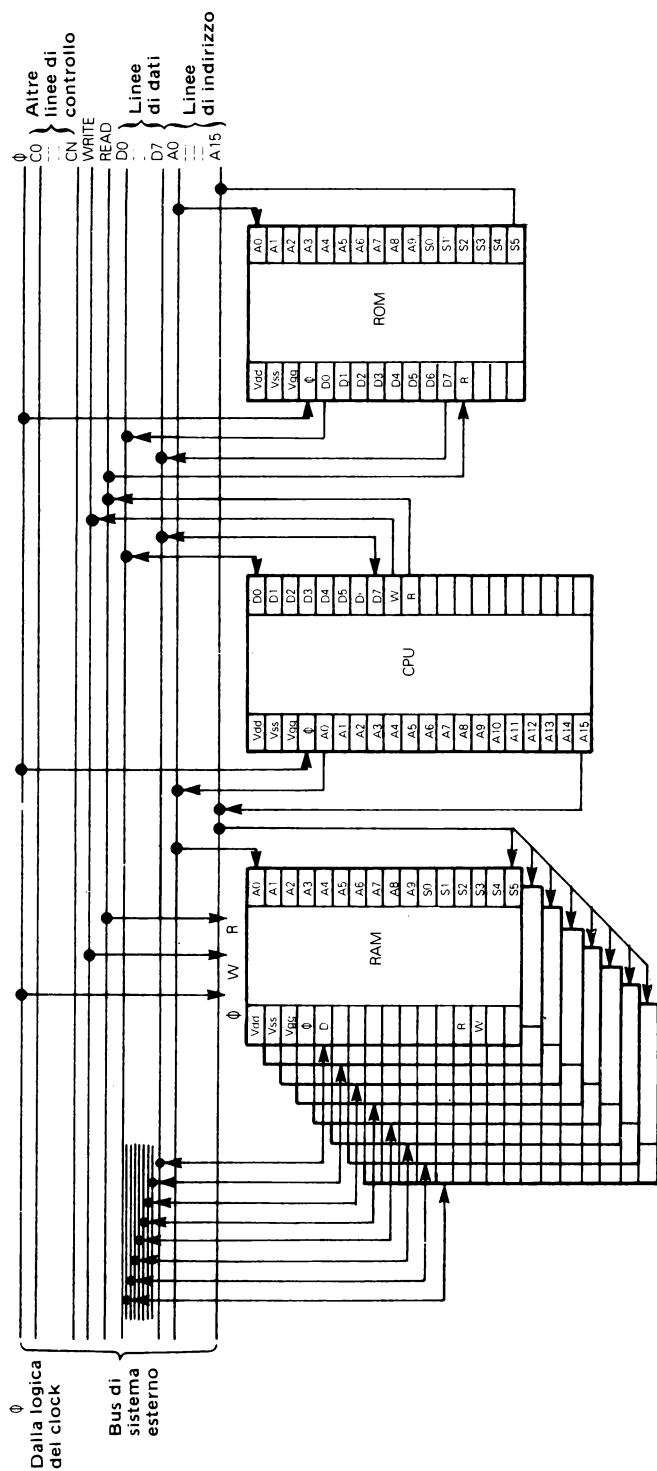



Figura 5-4. RAM (senza interfaccia RAM), ROM e CPU collegate tramite Bus Dati esterno

Figura 5-5. Interfaccia RAM, ROM e CPU collegate tramite Bus Dati esterno

quali ha lo stesso codice di selezione dispositivo, ma è collegato ad una diversa linea dati sul bus del sistema esterno. La Figura 5-3 illustra un solo chip RAM con logica di selezione dispositivo sul chip, e la Figura 5-4 mostra uno dei modi con cui si può aggiungere la memoria RAM alla combinazione ROM-CPU illustrata in Figura 5-2.

Alcuni microcomputer hanno speciali dispositivi logici di interfaccia della RAM. Questi dispositivi possono rinfrescare le RAM dinamiche o fornire logica di selezione dispositivo. Il Fairchild F8 ha bisogno di speciali dispositivi di interfaccia della RAM a causa della sua particolare distribuzione della logica. La Figura 5-5 illustra la RAM controllata da un dispositivo di interfaccia della RAM stessa.

IL TRASFERIMENTO DATI ALL'ESTERNO DEL SISTEMA MICROCOMPUTER (INPUT/OUTPUT)

Al trasferimento dei dati fra la logica che è parte del sistema microcomputer e la logica che ne è al di fuori, si fa in genere riferimento denominandolo input/output (I/O).

CONFINI DEL SISTEMA MICROCOMPUTER

Entro i confini dei sistemi microcomputer includeremo tutta la logica che è stata specificatamente progettata per operare insieme con la CPU. Classificheremo tutta l'altra logica come esterna.

L'interfaccia fra il microcomputer e la logica esterna dev'essere definita in modo chiaro, deve contenere tutto ciò che occorre per il trasferimento dei dati, più i segnali di controllo che identificano gli eventi man mano che hanno luogo.

Vi sono molti modi di eseguire il trasferimento dei dati fra il microcomputer e la logica esterna, ma essi ricadono tutti nelle tre categorie seguenti:

1) **I/O PROGRAMMATO.** In questo caso, tutti i trasferimenti di dati fra i microcomputer e la logica esterna sono completamente controllati dal microcomputer o, più precisamente da un programma che viene eseguito dalla CPU del microcomputer stesso.

   Vi sarà un protocollo ben definito per cui il microcomputer mette in evidenza che i dati messi in uscita sono stati messi in una posizione alla quale la logica esterna può avere accesso; o, in alternativa il microcomputer indicherà in un certo modo prestabilito che è in attesa che la logica esterna metta i dati in qualche posizione predefinita dalla quale essi possono essere prelevati dal microcomputer.
   La caratteristica chiave dell'I/O programmato è che la logica esterna fà quello che le viene detto di fare.

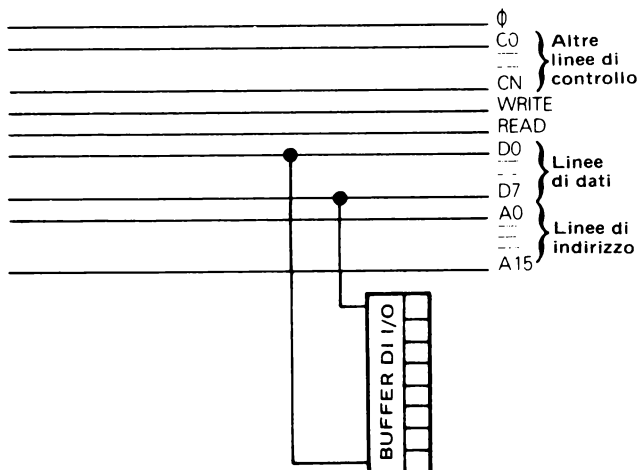
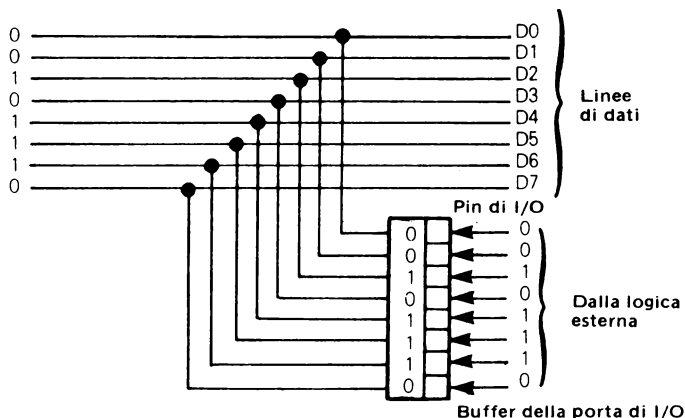
2) **I/O PER INTERRUPT.** Gli interrupt sono un mezzo a disposizione della logica esterna per forzare il microcomputer e sospendere qualunque cosa esso stia facendo in quel momento per rispondere ai bisogni della logica esterna.

3) **ACCESSO DIRETTO IN MEMORIA. (Direct Memory Acces, DMA).** Questa è una forma di trasferimento dati che permette loro di spostarsi fra la memoria del microcomputer e i dispositivi esterni senza coinvolgere la CPU nella logica del trasferimento dei dati.
   Le richieste fisiche per ogni tipo di I/O verranno descritte a suo tempo.

I/O PROGRAMMATO

I dati vengono trasferiti fra il microcomputer e la logica esterna, in entrambe le direzioni attraverso una porta di I/O.

PORTE DI I/O

Una porta di I/O consiste di un buffer di I/O, collegato con le linee dati del bus del sistema esterno e con i pin che accedono alla logica esterna:

Quando la logica esterna trasmette i dati al microcomputer, lo fa presentando i dati al pin della porta di I/O, da dove i dati vengono memorizzati nel buffer di I/O. Il valore binario 0011001110 verrebbe trasmesso nel modo seguente:

Il buffer della porta di I/O non può essere costantemente in comunicazione con le linee dati del bus del sistema esterno, come precedentemente illustrato, dato che può darsi che queste contengono dati da e verso la memoria. Se la porta di I/O comunicasse permanentemente con le linee dati del bus del sistema esterno, ogni volta che la logica esterna presenta dei dati ai pin I/O, questi dati sarebbero propagati lungo le linee dati in comune con conseguenze imprevedibili.

La CPU del microcomputer selezionerà quindi una portata di I/O e leggerà il contenuto del bus della porta di I/O, nello stesso modo in cui i dati vengono letti dalla me-

moria. Questo parallelo fra la lettura dei dati dai buffer delle porte di I/O e la lettura dei dati dalla memoria è appropriato, dato che la maggior parte dei microcomputer trasferiscono un grosso numero di dati da e verso la logica esterna; perciò, essi hanno più di una porta di I/O.

PORTE I/O INDIRIZZATE USANDO LE LINEE DI INDIRIZZAMENTO DI MEMORIA

Possiamo sviluppare un dispositivo di I/O parallelo con una o più porte di I/O, dove i buffer delle porte di I/O hanno degli indirizzi, esattamente come le parole di memoria. Ci dovrebbe essere un semplice sistema per prendere la linea di indirizzamento di ordine superiore (A15) e progettare la logica del microcomputer in modo che in qualunque momento questa linea sia, o venga selezionato un modulo di memoria. In altre parole, gli indirizzi di memoria $7FFF_{16}$ e inferiori avranno accesso alle parole di memoria, mentre gli indirizzi di memoria 8000_{16} e superiori avranno accesso ai buffer delle porte di I/O. Usando le linee di controllo READ e WRITE, la Figura seguente illustra un dispositivo di I/O parallelo con una porta.

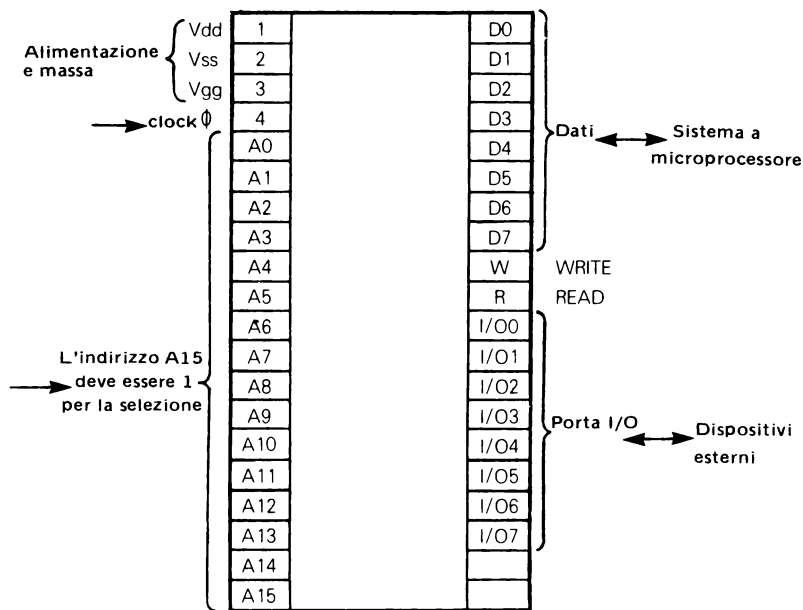



Figura 5-6. Dispositivo di interfaccia parallelo a una porta

Il dispositivo della figura è considerato un dispositivo di I/O parallelo perchè i dati sono scritti e letti in blocchi di 8 unità binarie, simultaneamente.

Notate che non vi sono ragioni per cui un dispositivo di I/O dovrebbe avere solo una porta di I/O. Un numero di porte di I/O che un dispositivo di I/O in parallelo ha, è semplicemente una funzione del numero di pin che sono economicamente disponibili su di un dual in-line package. Il dispositivo illustrato nella figura precedente usa i suoi pin in questo modo:

1) Sedici pin sono collegati alle linee di indirizzamento del bus del sistema esterno e forniscono le informazioni necessarie per determinare se il buffer di questa porta di I/O è stato selezionato.
2) Otto pin sono collegati alle linee dati del bus del sistema esterno e si usano per trasferire le informazioni dal bus del sistema esterno al buffer della porta di I/O o viceversa.
3) Due linee di controllo, READ e WRITE determinano se i dati andranno dal buffer della porta I/O verso il buffer del sistema esterno (lettura), o viceversa (scrittura).
4) Tre pin sono necessari per alimentazione e massa.
5) Un pin è necessario per il segnale di clock.

Si arriva cosi ad una somma di trenta pin, che lascia solo dieci pin disponibili per le porte di I/O. Perciò, il dispositivo di I/O in parallelo illustrato nella figura precedente può fare da supporto ad una sola porta di I/O.

Naturalmente, c'è un sistema ovvio per aumentare il numero di porte di I/O sul nostro dispositivo di I/O parallelo. Il fatto di avere sedici linee di indirizzamento implica il fatto che il microcomputer può indirizzare 32768 (2^{15}) porte di I/O e ciò è certamente troppo.

Cosa ne dite di ridurre a dieci il numero delle linee di indirizzamento? Ora sono disponibili 16 pin per le porte di I/O e il nostro dispositivo di I/O in parallelo può avere due porte di I/O, come illustra la figura seguente:

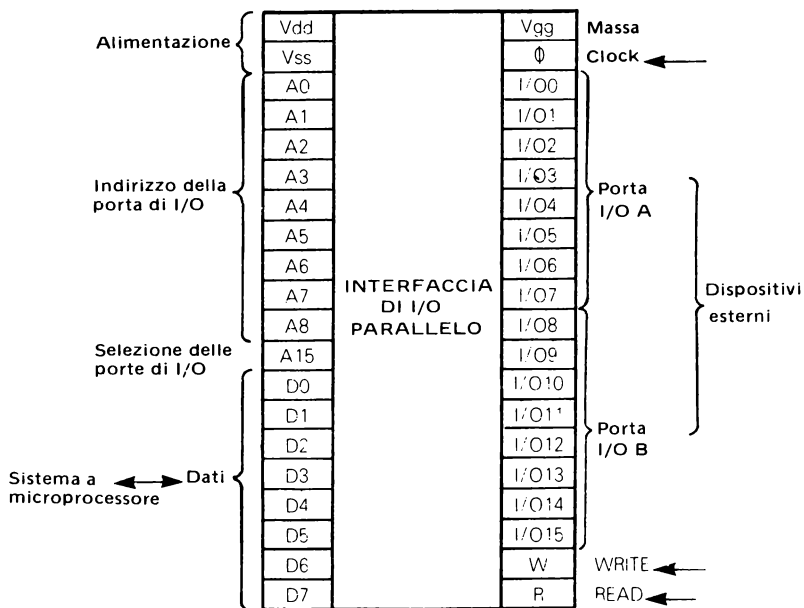
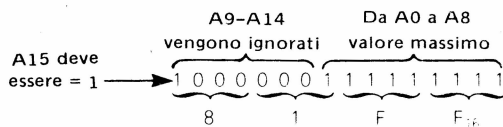



Figura 5-7. Chip di interfaccia di I/O parallelo a due porte

Le dieci linee di indirizzamento di questa figura saranno divise nel modo seguente:
1) Una delle dieci linee di indirizzamento sarà A15, dato che questa linea deve essere 1 per selezionare le porte di I/O anzichè una memoria.

2) Le nove linee di indirizzamento rimanenti possono essere nove delle altre quindici linee di indirizzamento. Per esempio, se sono collegate le linee di indirizzamento da A0 ad A8, gli indirizzi da 8000_{16} a $81FF_{16}$ selezioneranno le porte di I/O:

Nel caso improbabile che questo sistema non fornisca indirizzi sufficienti per le porte di I/O, le linee di indirizzamento da A9 a A14 potrebbero essere incluse come parte dell'indirizzo trasmesso ad un altro dispositivo di I/O in parallelo.

Quando si usa la linea di indirizzamento di ordine superiore (A15) per selezionare i buffer delle porte di I/O, ci si rimette per il fatto che possano essere indirizzate solo 32768(2^{15}) parole di memoria invece di 65536(2^{16}). Nel mondo dei minicomputer, questa riduzione di memoria indirizzabile può essere un caro prezzo da pagare per semplificare solo la selezione del buffer delle porte di I/O. **Nel mondo dei microcomputer, tale riduzione non è molto significativa dato che pochissimi microcomputer richiedono 32768 parole di memoria indirizzabile (o anche solamente un numero di parole dello stesso ordine di grandezza).** Di solito un'applicazione di un microcomputer richiede in totale un numero di parole di memoria variabile dalle 1024 alle 4096.

INDIRIZZI DELLE PORTE DI I/O

Ciononostante, molti microcomputer usano una logica di indirizzamento separata per selezionare le porte di I/O. La Figura 5-8 illustra un possibile schema che aggiunge due linee di controllo alla CPU del microcomputer. Una linea di controllo, IOSEL, specifica che le linee di indirizzamento da A0 a A7 contengono il codice di selezione del buffer della porta di I/O. L'altra linea di controllo, IORW, se alta, indica che il bus dati esterno contiene informazioni che devono essere lette nel buffer della porta di I/O. Se IORW è bassa, i contenuti selezionati del buffer della porta di I/O devono essere messi in output sulle linee dati del bus dati esterno.

Come scoprirete nel Capitolo 7, i due metodi che abbiamo descritto per la selezione delle porte di I/O sono solo due di una lunghissima serie di possibilità. Comunque, questi due metodi costituiscono i metodi comuni con cui vengono indirizzate le porte di I/O*.

Sfortunatamente, il trasferimento cieco di dati fra un microcomputer e la logica esterna non fornirà sempre sufficiente capacità di I/O. Vengono a mancare le seguenti funzioni:

1) **Il microcomputer deve essere in grado di dire alla logica esterna quando i dati sono stati posti in un buffer di I/O e sono pertanto pronti per essere prelevati. Viceversa, la logica esterna deve avere i mezzi per indicare al microcomputer che ha posto i dati in un buffer di I/O e i dati possono ora essere letti.**

2) **Il microcomputer e la logica esterna devono essere entrambi un modo di informare l'altro circa la natura dei dati posti in un buffer di I/O.**
   Cniaramente i dati che vengono trasferiti tra il microcomputer e la logica esterna sono soggetti a varie interpretazioni. Per esempio, possono essere dati numerici puri; ma possono anche essere un codice che identifica le operazioni da eseguire, o già eseguite. Possono essere anche una parte, o tutto un indirizzo.

CONTROLLO DI I/O

Quando il microcomputer manda segnali in output alla logica esterna come mezzo di identificare eventi o dati, questi segnali vengono definiti come controlli di I/O. La stessa informazione che viaggia in direzione opposta cioè dalla logica esterna al microcomputer, è chiamata stato di I/O. La differenziazione delle informazioni in controlli e stati, basata sulla direzione delle informazioni, è logica, dato che abbiamo a che fare con una situazione in cui il computer controlla in ogni momento gli eventi.

STATO DI I/O

In altre parole, il microcomputer mette in output controlli per controllare le sequenze della logica esterna. La logica esterna non può controllare il microcomputer; può solo mandare in input informazioni di stato perchè il microcomputer interpreti quando e come essa stia operando.

I sistemi minicomputer hanno solitamente un intero set di linee di stato e di controllo di I/O che sono separate e distinte dalle porte di I/O. I microcomputer allocano più comunemente una o più porte di I/O perchè funzionino come canali di stato e di controllo, mentre le altre porte di I/O trasferiscono i dati.

La porta A di I/O nella Figura 5-8 potrebbe essere usata per trasferire dati, mentre la porta B di I/O potrebbe essere usata per trasferire informazioni di stato e di controllo. Finchè si ha a che fare con i microcomputer, si usano le stesse sequenze di istruzioni per trattare il flusso dati attraverso entrambe le porte di I/O. E' il modo in cui il microcomputer interpreta una parola che determina se la parola è un dato, o un'informazione di stato o di controllo.

I/O PER INTERRUPT

La maggior parte delle CPU dei microcomputer hanno un segnale di controllo per mezzo del quale la logica esterna può richiedere l'attenzione del microcomputer. Si fa riferimento a questo segnale come ad un segnale di richiesta d'interruzione e di interrupt perchè, in effetti, la logica esterna domanda al microcomputer di interrompersi, qualunque cosa esso stia facendo in quel momento, allo scopo di soddisfare le sue necessità più urgenti.

IL CONCETTO DI INTERRUPT

Inizieremo a parlare delle interruzione con un esempio che è troppo semplice per avere riscontro nella realtà, ma che contiene tutte le caratteristiche chiave di un'applicazione significativa.

Supponiamo che un microcomputer venga usato per controllare la temperature dell'acqua della doccia, come mostra la Figura 5-9. Un termometro misura la temperatura dell'acqua calda e fredda mescolata che esce dal rubinetto della doccia e trasmette questa temperatura, come segnale digitale, al microcomputer. Il microcomputer confronta la temperatura con un valore prestabilito, che è fornito da un appropriato controllo. A seconda della differenza fra la temperatura reale della doccia e quella desiderata, il microcomputer mette in output dati che devono essere interpretati come un segnale di controllo della valvola, facendo in modo che la valvola aumenti o diminuisca il flusso dell'acqua calda.

Vi sono dei motivi per cui questa applicazione che sembra semplice in realtà non lo è affatto. Come l'esperienza vi avrà insegnato vi è un ritardo fra il momento in cui sistemate il rubinetto della doccia e quello in cui l'acqua che esce della doccia stessa cambia temperatura. Per questo motivo, **il microcomputer dovrà eseguire un programma ben fatto per essere sicuri che non tenti di fare delle variazioni ridicole. Chiameremo**

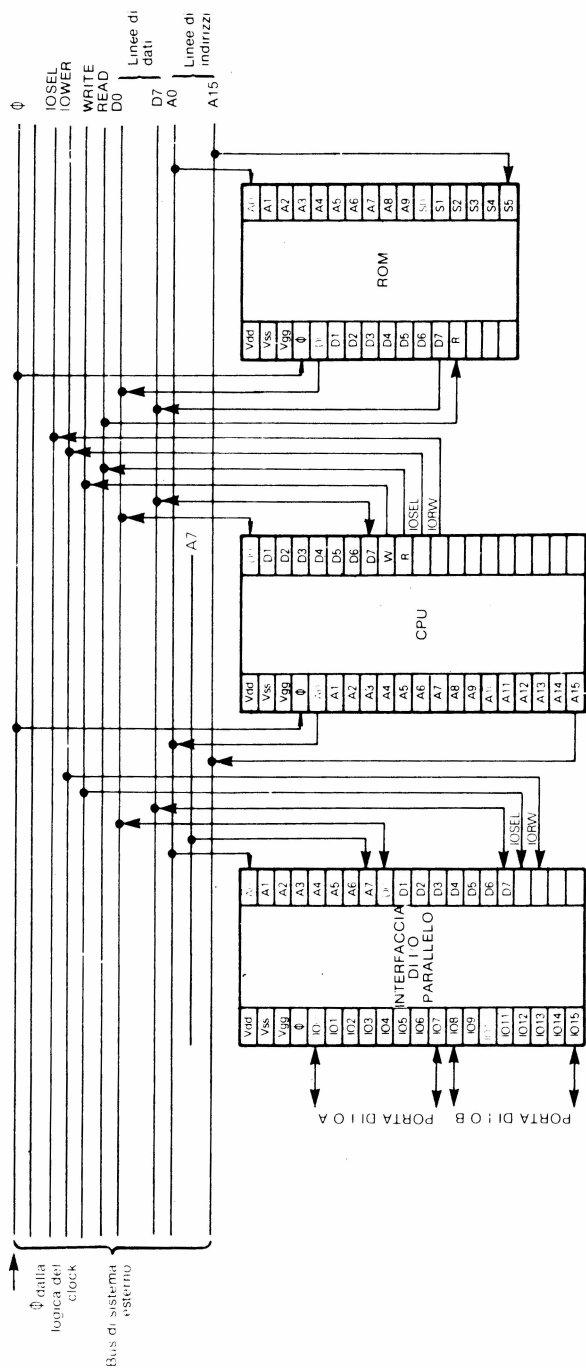
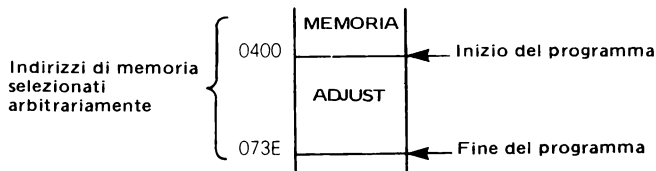
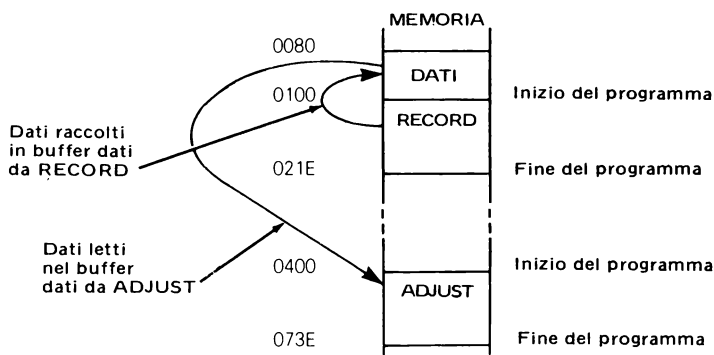



Figura 5-8. Chip di interfaccia di I/O parallelo che utilizza una logica di indirizzamento di I/O

questo programma ADJUST, e diremo che risiede in memoria di programma a questi indirizzi:

Un altro programma, chiamato RECORD, prenderà in input i dati dal lettore di temperatura, interpretandoli correttamente per poi rappresentare la lettura della temperatura. L'unico contatto fra i programmi RECORD e ADJUST avviene in quanto ADJUST trova in anticipo i dati in una certa area di memoria dati e RECORD mette i dati nel formato corretto, in quell'area richiesta. La nostra memoria appare ora in questo modo:

COME TRATTARE UNA RICHIESTA DI INTERRUZIONE

Il modo in cui le temperature della doccia vengono lette e trasmesse al microcomputer è un altro aspetto di questo problema che non è così chiaro e semplice come potrebbe sembrare. Un lettore di temperatura non costoso impiegherà circa mezzo secondo per registrare una temperatura. Mezzo secondo può non sembrare un periodo di tempo lungo, ma un microcomputer può eseguire circa un quarto di milione di istruzioni in questo periodo di tempo.

Come fa il microcomputer a sapere quando il lettore della temperatura ha un nuovo valore da trasmettere? Se il lettore della temperatura tenta semplicemente di mandare i dati ad una porta di I/O, è molto probabile che il microcomputer non riesca a leggere la temperatura. Una lettura può andare facilmente perduta nell'esecuzione di un quarto di milione di istruzioni.

RICHIESTA D'INTERRUZIONE

Un modo di risolvere il problema è illustrato nella Figura 5-10. **Una sequenza in tre passi permette al lettore di temperature di richiamare l'attenzione del microcomputer in questo modo.**

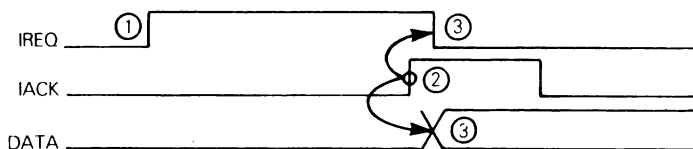
① **Il lettore di temperatura trasmette un segnale di richiesta di interruzione (IREQ) al microcomputer attraverso una linea di controllo del bus del sistema esterno.**

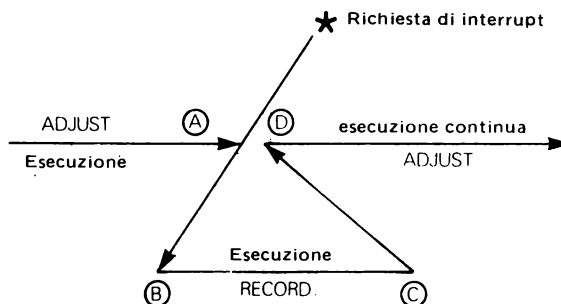
Figura 5-9. Controllo a microprocessore della temperature dell'acqua in una doccia

RICONOSCIMENTO DELL'INTERRUPT

② **Il microcomputer ha la scelta di accettare o di rifiutare la richiesta di interruzione; esso accetta la richiesta d'interruzione mettendo in uscita un segnale di riconoscimento di interruzione (IACK) sulla linea di controllo del bus del sistema esterno.**

③ **Il dispositivo esterno usa il segnale di riconoscimento di interruzione come un segnale di abilitazione, per trasmettere i dati alla porta di I/O A. Inoltre, il dispositivo esterno deve eliminare il suo segnale di richiesta di interruzione quando riceve un riconoscimento d'interruzione, dato che, chiaramente dopo la richiesta d'interruzione è stata servita, il dispositivo esterno non richiede più un'altra interruzione.**

Il Timing per questa sequenza in tre passi si può illustrare come segue:

Notate che, sebbene abbiamo parlato del dispositivo interno che inserisce i dati nel microcomputer, il flusso dei dati potrebbe avvenire altrettanto facilmente nella direzione opposta; infatti, non vi è ragione per cui un qualsiasi flusso dati abbia bisogno di seguire un'interruzione. Il programma eseguito seguendo un'interruzione potrebbe, ad esempio, mettere semplicemente in output dei segnali di controllo.

Lo scopo dell'interrupt è di dire al microcomputer che esso deve sospendere qualunque cosa stia facendo, elaborare i dati che vengono inseriti, poi proseguire con le sue operazioni precedentemente sospese. Prendendo come riferimento i programmi RECORD e ADJUST, ecco che cosa viene:

Facciamo ancora riferimento alla Figura 5-10. I passi ① e ② fanno si che la richiesta d'interruzione venga letta dal microcomputer ad Ⓐ . La CPU del microcomputer risponde sospendendo l'esecuzione di ADJUST, ed esegue RECORD (da Ⓑ a Ⓒ). Mentre viene eseguito RECORD, i dati trasmessi dal lettore di temperatura ③ nella Figura 5-10 vengono letti nel microcomputer, dato che il programma RECORD è stato scritto per anticipare l'arrivo di questi dati.

Quando RECORD ha completato l'esecuzione a Ⓒ , continua l'esecuzione di ADJUST a Ⓓ , riprendendo esattamente da dove si era interrotta ad Ⓐ

Figura 5-10. Dispositivo esterno che utilizza una richiesta di interrupt per comunicare al microprocessore che i dati possono essere introdotti

La caratteristica chiave dell'esecuzione del programma RECORD è che è un evento non programmato. Non vi è nessuna logica all'interno del microcomputer che possa prevedere quando o con che frequenza verrà eseguito il programma RECORD. Comunque, vi è una logica all'interno del microcomputer che può sospendere l'esecuzione di un qualsiasi programma, riprendendo l'esecuzione più tardi dal punto della sospensione.

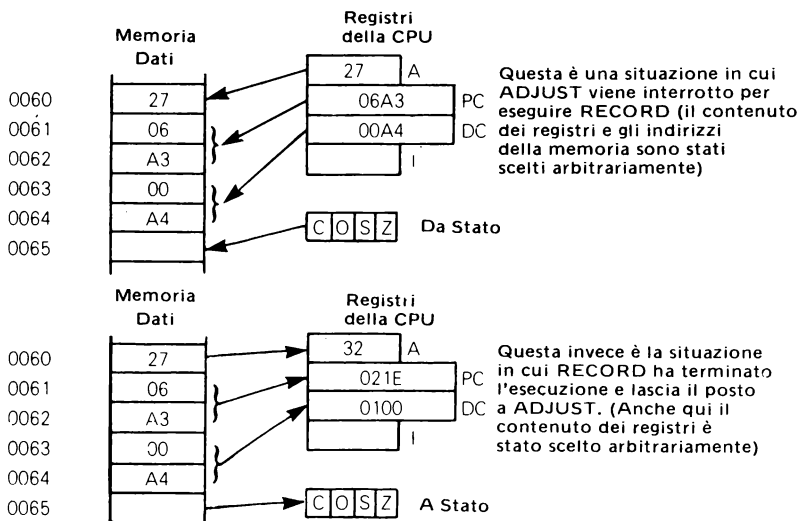
LA RISPOSTA DEL MICROCOMPUTER AD UN INTERRUPT

Al suo livello più elementare la CPU di un microcomputer potrebbe rispondere ad una richiesta di interruzione caricando semplicemente nel Program Counter l'indirizzo di inizio del programma che la logica esterna di interruzione vuole venga eseguito. Ma questo pone due problemi:

1) **Che cosa succede al programma che era in esecuzione?**
2) **Da dove ottiene la CPU l'indirizzo del programma che la logica di interruzione vuole venga eseguito?**

Consideriamo prima che cosa succede al programma che era in esecuzione.

SALVATAGGIO DEI REGISTRI E DELLO STATO

Il programma vecchio può avere informazioni importanti nei flag di stato, nel data counter e nell'accumulatore; questi dati stanno per essere distrutti dal programma nuovo, così che quando avrà finito l'esecuzione il programma vecchio non sarà più in grado di ricominciare. Questo problema viene risolto conservando il contenuto di tutti i registri della CPU, compreso il Program Counter prima di iniziare l'esecuzione del nuovo programma. Quando l'esecuzione del nuovo programma termina, il valore precedentemente salvato del Program Counter è l'indirizzo dell'istruzione che stava per essere eseguita quando il vecchio programma è stato interrotto; così, semplicemente ripristinando i valori dei registri della CPU, il vecchio programma può riprendere dal punto in cui era stato sospeso. Questo concetto è illustrato nella maniera seguente:

Un interrupt non sarà riconosciuto finchè l'istruzione in corso non avrà completato l'esecuzione. In questo caso, non c'è bisogno di conservare il contenuto del registro istruzioni, dato che esso contiene un codice istruzione che è già stato elaborato. In altre parole, l'interruzione precede direttamente l'arrivo di un nuovo codice istruzione.

Vi sono due modi di conservare i flag di stato e il contenuto dei registri del vecchio programma come precedentemente illustrato, prima che il nuovo programma incominci l'esecuzione. Un modo sarebbe quello di far iniziare al segnale di richiesta interruzione l'esecuzione di un microprogramma (memorizzato nell'unità di controllo) che scriva semplicemente il contenuto dei registri della CPU in una area dati della memoria che è stata riservata per questo scopo. Il progettista di microcomputer può essere restio a sprecare in questo modo dello spazio prezioso nell'unità di controllo, e può invece richiedere al programmatore di scrivere un breve programma che faccia la stessa cosa. Un programma di questo tipo si chiama routine di servizio dell'interrupt.

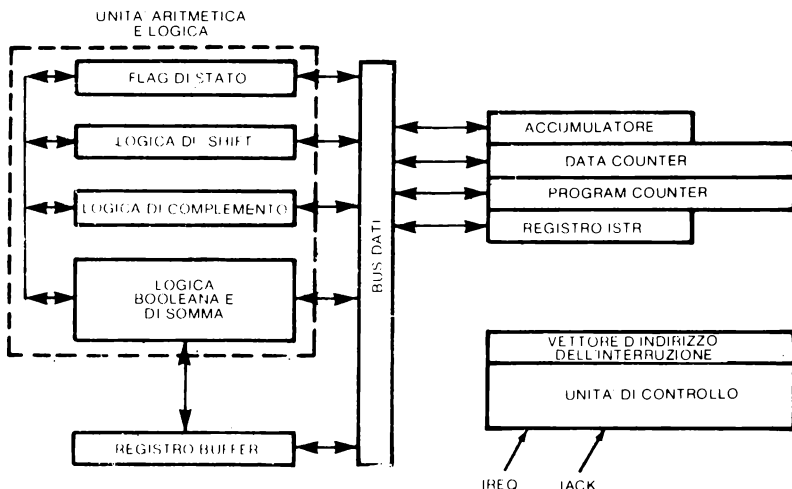
ROUTINE DI SERVIZIO DELL'INTERRUPT

Alla fine dell'esecuzione del nuovo programma qualunque sia la logica usata per conservare il contenuto dei registri del vecchio programma deve essere eseguita l'operazione contraria, per ripristinare i registri e lo stato del vecchio programma.

Parleremo più dettagliatamente della routine di servizio dell'interrupt nel Capitolo 6, dopo avere discusso nei dettagli della programmazione.

VETTORE D'INDIRIZZO DELL'INTERRUZIONE

Consideriamo ora come la CPU del microcomputer ottiene l'indirizzo del programma che la logica di di interruzione vuole venga eseguito. Faremo riferimento a questo particolare indirizzo come al vettore di indirizzo dell'interruzione.

I modi per determinare quale programma deve essere eseguito dopo una richiesta di interruzione sono tanti quasi quanti i microcomputer. Nel caso del nostro problema di controllo della temperatura della doccia, vi è una soluzione semplice. Il lettore della temperatura della doccia è l'unico dispositivo esterno che può richiedere una interruzione, e vi è un solo programma di cui può volere l'esecuzione dopo l'interruzione. In questo caso, la CPU del microcomputer potrebbe essere costruita con una logica interna che fa sì che, dopo l'interruzione venga eseguito un programma, che ha origine ad uno specifico indirizzo di memoria:

Ogni volta che l'unità di controllo riceve una richiesta d'interruzione (IREQ) ed è pronta per servire l'interruzione stessa, fa così:

1) Emette il segnale di riconoscimento dell'interruzione, IACK.

2) Salva il contenuto dei flag di stato, dell'accumulatore, del data-counter e del Program Counter o dar modo al programmatore di fare la stessa cosa.

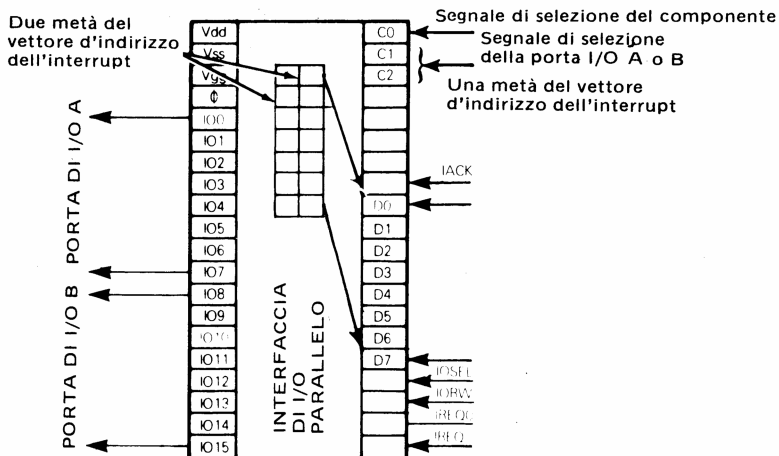
3) Sposta il contenuto del vettore di indirizzo dell'interruzione nel Program Counter.

Il programmatore di un minicomputer considererebbe questo metodo di risposta alle interruzioni ridicolo. Chissà quando e come potrà rispondere in seguito ad una nuova interruzione? Specificare che tutte le interruzioni saranno servite da un programma originato all'indirizzo di memoria 0400_{16} (per esempio) sarebbe una limitazione intollerabile, perchè limita fortemente il programmatore togliendogli ogni flessibilità.

Nelle applicazioni dei microcomputer, il fatto di avere vettori di indirizzo dell'interruzione fissi, ha un certo senso. Ricordate che i microcomputer stanno per essere usati come componenti logici, non come computer universali. La maggior parte dei microcomputer vengono usati in modo dedicato, senza, variazioni, dove una o alcune interruzioni specifiche richiederanno risposte ugualmente specifiche.

La Figura 5-11 mostra come si potrebbero modificare il dispositivo di interfaccia di I/O in parallelo e il dispositivo ROM della Figura 5-8 per ricevere il segnale di richiesta di interruzione IREQ. Ogni dispositivo, come è modificato nella Figura 5-11, contiene un registro di 16 bit, nel quale è permanentemente memorizzato un vettore di indirizzo dell'interruzione. Quando riceve il segnale di riconoscimento di interruzione IACK, il dispositivo trasmette il vettore di indirizzo dell'interruzione alla CPU, per mezzo delle linee di indirizzo del bus del sistema esterno. Dopo aver messo da parte il contenuto dei flag di stato e dei suoi registri, la CPU carica il vettore d'indirizzo dell'interruzione nel Program Counter.

LINEE MULTIPLEXED

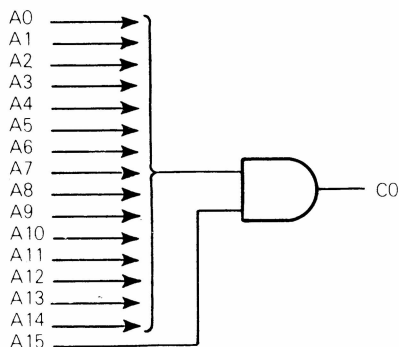
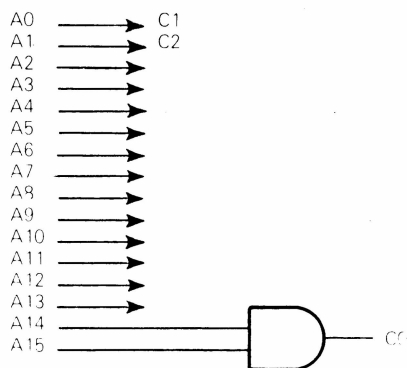
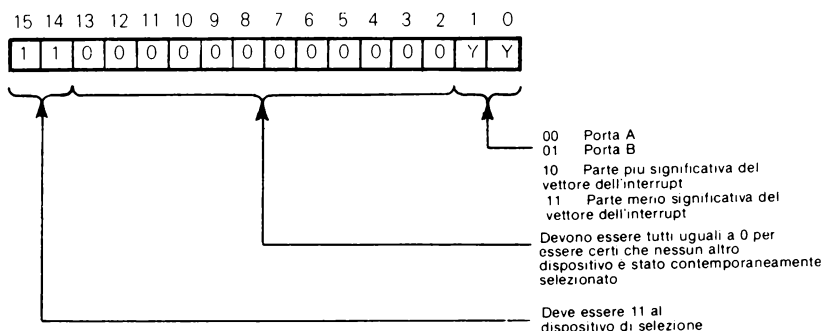
Il dispositivo d'interfaccia di I/O in parallelo, avendo solo otto pin di indirizzo collegati ad A0-A7, dovrà trasmettere un vettore d'indirizzo dell'interruzione alla CPU in due parti e il microprogramma dell'unità di controllo della CPU dovrà caricare ogni metà in modo appropriato nel Program Counter. A questo si fa riferimento come a linee multiplexed, cioè, l'uso delle stesse linee del bus per portare segnali che devono essere interpretati in modi diversi, in tempi diversi.

Notate che abbiamo un problema con il dispositivo d'interfaccia di I/O in parallelo: come abbiamo visto, abbiamo esaurito i pin. Bisogna in qualche modo prendere in input lo IACK in risposta all'IREQ0.
Il modo migliore di risolvere questo problema è sostituire i pin d'indirizzo da A0 ad A7 con tre pin di selezione del chip C0, C1 e C2. Restano così cinque pin liberi, uno dei quali può essere assegnato allo IACK.

Il dispositivo di I/O parallelo cambia in due modi importanti.

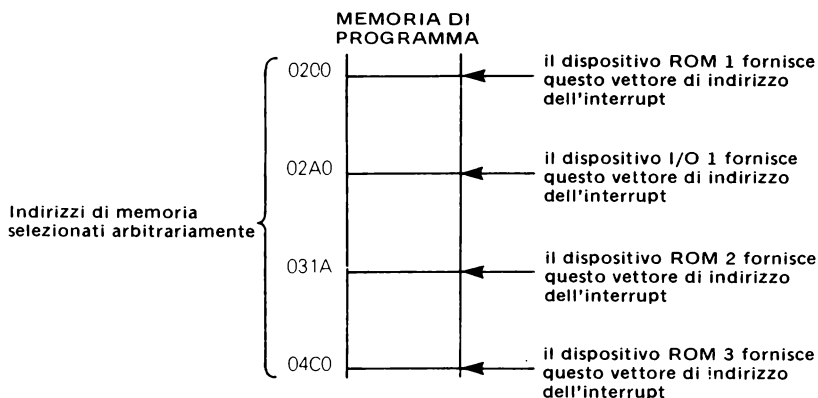
Primo C0, C1 e C2 saranno il prodotto della logica addizionale di selezione del chip, che riceve alcune o tutte le linee d'indirizzo del bus del sistema esterno come input.

Secondo, il bus dati è ora multiplessato; in momenti diversi, esso può trasmettere alla CPU dati o indirizzi.

SELEZIONE DEI DISPOSITIVI DI I/O

Negli attuali dispositivi di supporto dei microcomputer, l'avere una logica di selezione del chip esterna costituisce una regola piuttosto che un'eccezione. In realtà, la logica di selezione esterna spesso non esiste. Dato che vi sono pochissimi dispositivi di I/O in un tipico microcomputer, e più di 32 K byte di memoria sono rari, potreste creare come l'AND di A15 e di una qualunque altra linea d'indirizzo:

Due qualunque dalle 14 linee d'indirizzo libere possono essere legate a C1 e C2, al fine di selezionare una delle quattro posizioni indirizzabili sul dispositivo di interfaccia di I/O parallelo. Prendete in considerazione questo esempio:

Gli indirizzi di memoria possono selezionare le porte nel modo seguente:

Così $C000_{16}$ indirizza la porta A
$C001_{16}$ indirizza la porta B
$C002_{16}$ indirizza la parte più significativa del vettore d'interruzione
$C003_{16}$ indirizza la parte meno significativa del vettore d'interruzione

Multiplessare il bus dati non è un problema; significa semplicemente che la routine di servizio dell'interrupt — il programma eseguito immediatamente dopo il riconoscimento d'interruzione — deve leggere i contenuti delle posizioni di memoria $C002_{16}$ e $C003_{16}$ e trattare questi due byte dati come l'indirizzo d'inizio del programma che deve essere eseguito.

Il fatto che il segnale di richiesta di interruzione arrivi alla memoria o al dispositivo d'interfaccia, invece che alla CPU, significa che per ogni memoria o dispositivo d'interfaccia del sistema, un dispositivo esterno diverso può avere la sua propria routine di servizio interruzione identificata come eseguibile dopo un'interruzione. Questo concetto è illustrato come segue:

In questa illustrazione, i segnali di richiesta d'interruzione che arrivano ai dispositivi ROM 1 e 2 e 3 specificheranno sempre l'esecuzione dei programmi memorizzati agli indirizzi 0200_{16} $031A_{16}$ e $04C0_{16}$, rispettivamente? Un segnale di richiesta d'interruzione che arriva al dispositivo d'interfaccia di I/O in parallelo 1 specificherà sempre l'esecuzione del programma memorizzato all'indirizzo $02A0_{16}$. Ogni richiesta d'in-

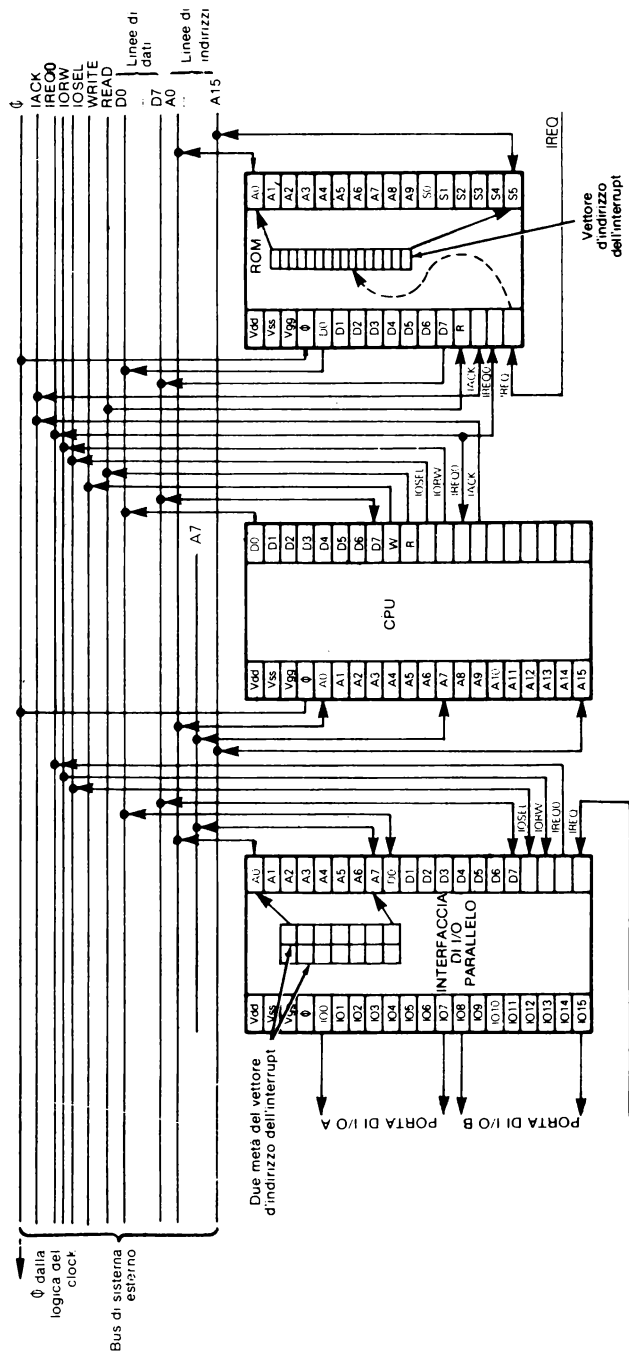



Figura 5-11. Uso di chip di I/O e ROM per la gestione degli interrupt

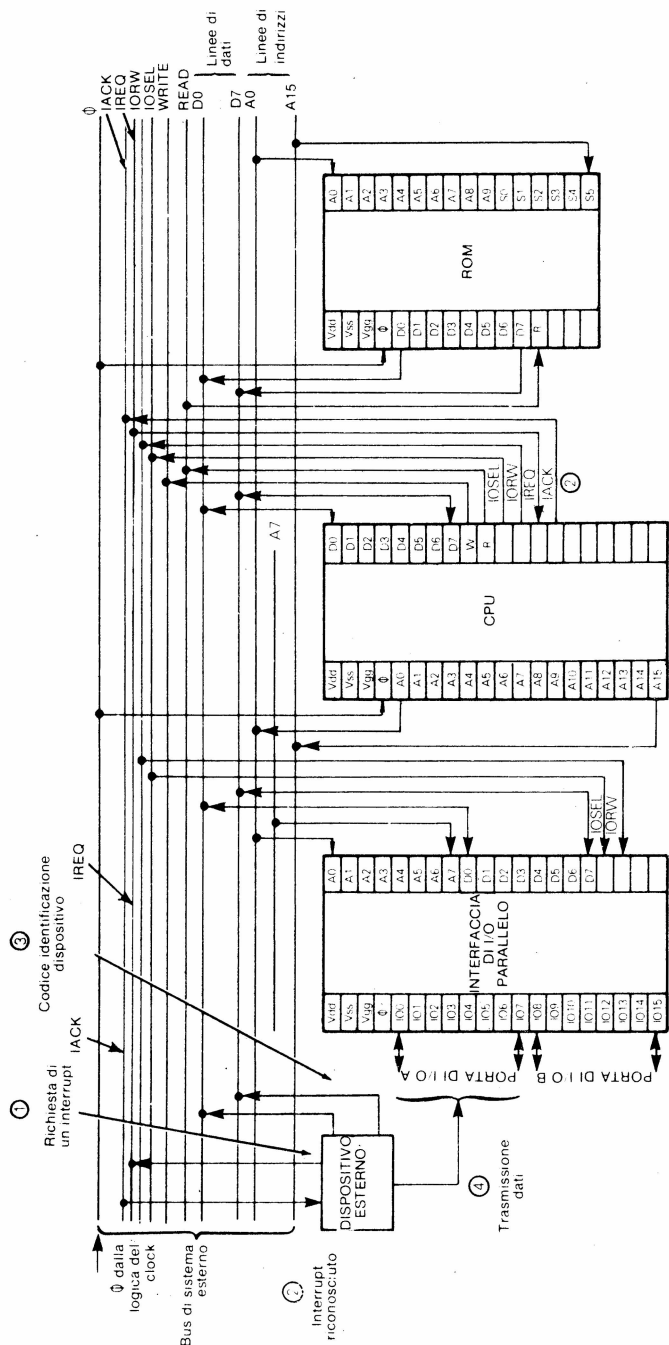



Figura 5-12. Dispositivo esterno che utilizza una richiesta di interrupt e codice di identificazione del dispositivo per comunicare al microprocessore che i dati possono essere introdotti.

terruzione arriva come un segnale IREQ separato e distinto; viene inoltrato alla CPU come IREQ0. Quando la CPU manda il segnale di riconoscimento IACK, la ROM o il dispositivo di I/O che richiede l'interruzione trasmettono il vettore d'indirizzo della interruzione, che è una caratteristica permanente di ogni ROM o del dispositivo di I/O.

Ma sorge un nuovo problema quando in un microcomputer sono compresi più dispositivi capaci di richiedere un'interruzione. Che cosa succede quando più dispositivi richiedono simultaneamente un'interruzione? Quale dispositivo deve essere riconosciuto, e come si evita il riconoscimento degli altri dispositivi?

La risposta è divisa in due parti. Primo, dobbiamo fornire ai dispositivi un mezzo per identificare se stessi; a questo scopo abbiamo usato i codici di selezione. Inoltre dobbiamo includere una logica che stabilisca la priorità dell'interruzione.
Prendiamo in esame questi due concetti.

CODICI DI SELEZIONE DISPOSITIVI D'INTERRUZIONE

Consideriamo un modello perfezionato dal nostro controllore di temperatura della doccia, progettato per controllare le temperature di molte docce per motel ed hotel. Un microcomputer è sicuramente abbastanza veloce per controllare le temperature di 100 e più docce.

Non sarebbe molto economico o pratico richiedere che vengano aggiunti al microcomputer una nuova memoria o un nuovo dispositivo di I/O per ogni nuova doccia da controllare.
Molti microcomputer richiederanno comunque un dispositivo esterno, quando vi è una richiesta d'interruzione, per accompagnare la richiesta con un codice di identificazione. La Figura 5-12 mostra come un dispositivo esterno può collegarsi con il bus del sistema esterno, mettendo il suo codice di identificazione sulle linee dati del bus del sistema esterno quando la CPU riconosce la sua richiesta d'interruzione con IACK.
Richiamandoci alla Figura 5-12, gli eventi procedono in questo modo:

1. La logica del dispositivo esterno crea un segnale di richiesta d'interruzione, che trasmette alla CPU come IREQ.
2. Quando la CPU è pronta per servire la richiesta d'interruzione; risponde mettendo in output uno IACK.
3. Quando riceve lo IACK, la logica del dispositivo esterno mette il codice di selezione di 8 bit sulle linee dati del bus del sistema esterno. La CPU riceve questi dati e li interpreta come codice di identificazione del dispositivo esterno.
4. Seguendo il protocollo specificato dal microcomputer, il dispositivo esterno mette i suoi dati ad una porta di I/O del dispositivo di interfaccia di I/O in parallelo.

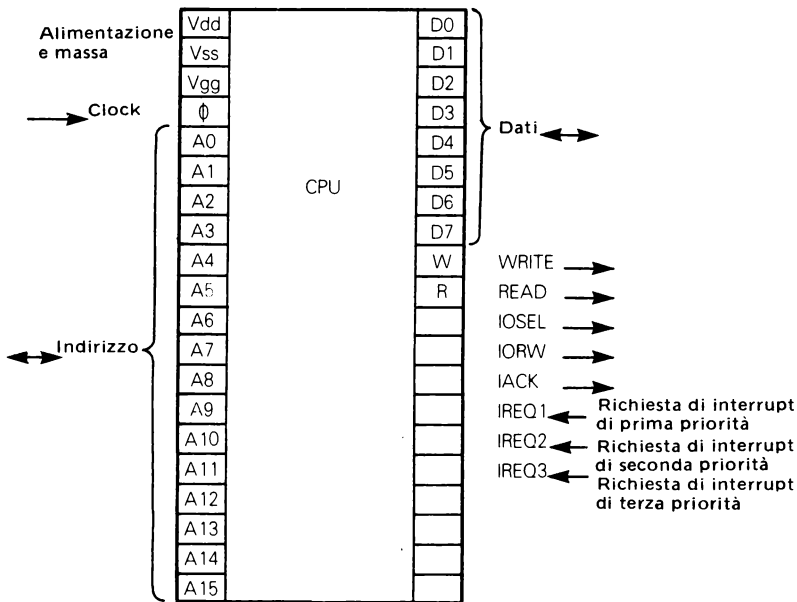
L'idea di avere dispositivi esterni che identifichino se stessi con un codice, come illustra la Figura 5-12, è molto significativa per un utente di minicomputer (o per il progettista), ma per l'utente di microcomputer ha una pecca chiarissima: richiede l'intelligenza del dispositivo esterno. Ricordate che un minicomputer può costare migliaia di dollari e può essere parte di un sistema che costa decine di migliaia di dollari. Pochissimi microcomputer costano più di 100 dollari, perciò possiamo solo permetterci l'uso di logica da pochi dollari per generare un codice di selezione dispositivi.

Ecco un'altra ragione per cui i minicomputer e i microcomputer sono e probabilmente resteranno, fondamentalmente diversi. Fornire i dispositivi esterni della logica indicata in Figura 5-12 può costare solo pochi dollari, il che è insignificante nel mondo dei minicomputer. Ma anche un dispositivo ROM, o un dispositivo di interfaccia di I/O parallelo costano solo pochi dollari. Tuttavia, ogni richiesta di intelligenza dedicata nei dispositivi esterni è, in confronto una richiesta costosa nel mondo dei micro-

computer. I codici di selezione dispositivi esterni, che sono così ovvi nel mondo dei minicomputer, sono una spesa che deve essere giustificata caso per caso nel mondo dei microcomputer. I costi dei chip aumentano molto poco all'aumentare della complessità, quindi l'economia richiede che il microcomputer faccia il più possibile, e richieda il meno possibile di logica esterna.

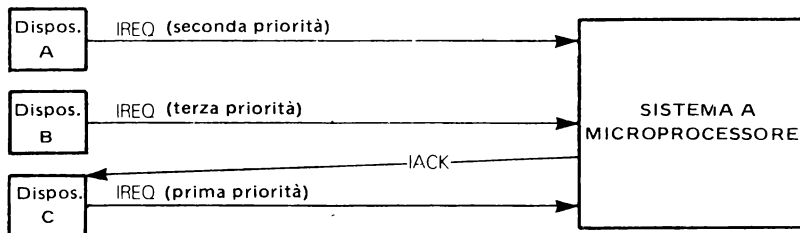
Vediamo un esempio facile. Due minicomputer costano 3250 e 3640 dollari. La scelta non facile per determinare quale minicomputer è realmente più costoso rende i due prezzi difficili da valutare. Se un minicomputer richiede che un dispositivo esterno abbia una logica extra del valore di 10 doliari per gestire gli interrupt, questa spesa extra avrà un'influenza limitata sull'economia generale.
Due microcomputer, configurati per un'applicazione ben specifica, costano 53 e 61 dollari rispettivamente. Se un dispositivo esterno avrà bisogno di logica extra per un valore di 10 dollari per richiedere un'interruzione sul microcomputer da 53 dollari, ma non sul microcomputer da 61 dollari, il microcomputer da 53 dollari può essere scartato per questa sola ragione.

PRIORITA' DEGLI INTERRUPT

Che cosa succede quando, nello stesso momento, più di un dispositivo esterno richiede un'interruzione? Questo problema può essere risolto in due modi. Primo, la logica all'interno del microcomputer può avere un numero di linee di richiesta d'interruzione con priorità ascendenti, così:

CHIP DI PRIORITA' DELL'INTERRUPT

Uno speciale dispositivo di priorità dell'interrupt potrebbe assolvere la stessa funzione, usando meno pin della CPU. Consideriamo il dispositivo di priorità d'interruzione illustrato in Figura 5-13.

PRIORITA' DELL'INTERRUPT E SUO SIGNIFICATO

Prima di spiegare come funziona il dispositivo di priorità dell'interrupt suddetto, definiremo che cosa si intende per priorità dell'interrupt. Supponiamo che un dispositivo esterno possa richiedere un'interruzione. Se due o più dispositivi esterni richiedono interruzioni mandando **simultaneamente** segnali IREQ che si sovrappongono nel tempo, allora il dispositivo esterno che ottiene il riconoscimento dell'interrupt (IACK) è determinato dalla priorità dell'interrupt.

In questa figura, i dispositivi esterni A, B e C richiedono tutti interruzioni trasmettendo simultaneamente segnali IREQ al microcomputer. Qualunque tecnica per la decisione dell'interruzione venga usata dal microcomputer si determina che il dispositivo C ha la priorità massima. Il dispositivo A ha la seconda priorità e il dispositivo B la priorità minima. L'unico segnale di riconoscimento, IACK, deve perciò essere mandato al dispositivo C.

Il fatto che le richieste d'interruzione dei dispositivi A e B non sono state riconosciute, non implica che essi debbano togliere i loro segnali di richiesta d'interruzione IREQ. Possono farlo se lo vogliono. Se non lo fanno, verranno riconosciuti, a turno,

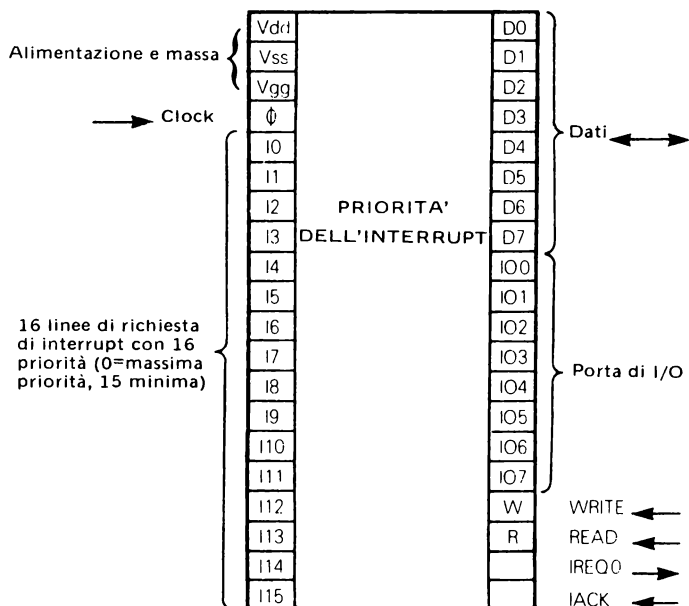
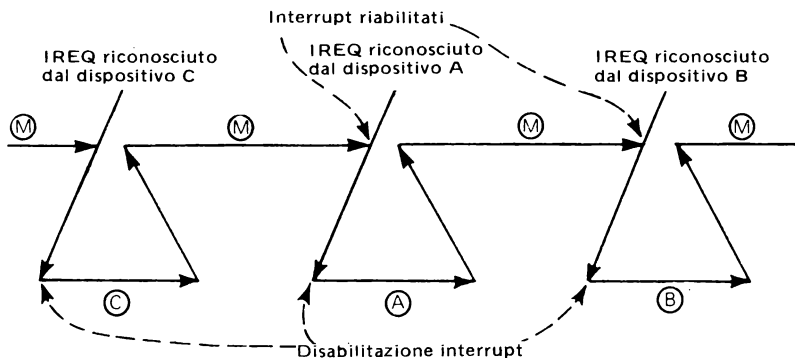



Figura 5-13. Dispositivo di priorità di interrupt

quando la CPU del microcomputer sarà successivamente pronta per riconoscere di nuovo gli interrupt.

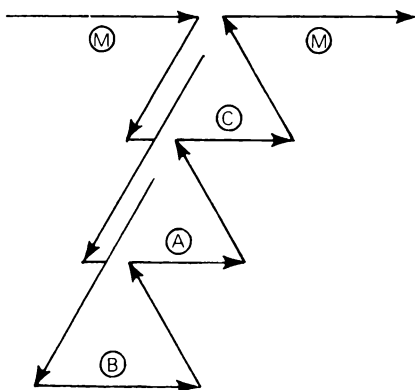
Facciamo riferimento ancora all'illustrazione dei dispositivi A, B e C, che richiedono tutti simultaneamente il servizio d'interruzione. In memoria vi sono tre programmi di servizio dell'interrupt, uno per ogni dispositivo. Questi programmi possono essere eseguiti in questo modo:

Ⓜ rappresenta il programma principale in esecuzione, e che viene interrotto. La routine di servizio dell'interrupt del primo dispositivo C viene eseguita a Ⓒ . In seguito le routine di servizio dell'interrupt dei dispositivi A e B vengono eseguite a Ⓐ e Ⓑ , rispettivamente.

Se le routine di servizio dell'interrupt Ⓒ , Ⓐ e Ⓑ devono essere eseguite sequenzialmente, come prima illustrato, mentre è in esecuzione Ⓒ , la logica che tratta le interruzioni del microcomputer deve essere disabilitata, così le richieste di interruzione (IREQ) dei dispositivi A e B sono ignorate. Ad un certo punto, dopo che Ⓒ ha completato l'esecuzione ed Ⓜ riprende ad essere eseguito, la logica dell'interrupt del microcomputer è riabilitata; ora l'IREQ del dispositivo A è riconosciuto. Mentre viene eseguita Ⓐ , la logica d'interruzione del microcomputer viene nuovamente disabilitata fino al momento in cui viene rimessa l'esecuzione di Ⓜ Dato che il dispositivo B sta ancora richiedendo un'interruzione, viene ora eseguita Ⓑ

Per abilitare e disabilitare la logica d'interruzione dei microcomputer, si usano speciali istruzioni. Queste istruzioni e il modo in cui vanno usate, sono descritte nel Capitolo 7. Supponiamo che il microcomputer non abbiá disabilitato la sua logica d'interruzione mentre eseguite le routine di servizio dell'interrupt Ⓐ , Ⓑ e Ⓒ . Il disegno di pagina successiva mostra come verrebbero servite le interruzioni.

Il concetto importante da capire è che le priorità dell'interrupt determinano quale dispositivo riceve il riconoscimento d'interruzione IACK quando più di un dispositivo richiede simultaneamente un'interruzione per mezzo di un IREQ.

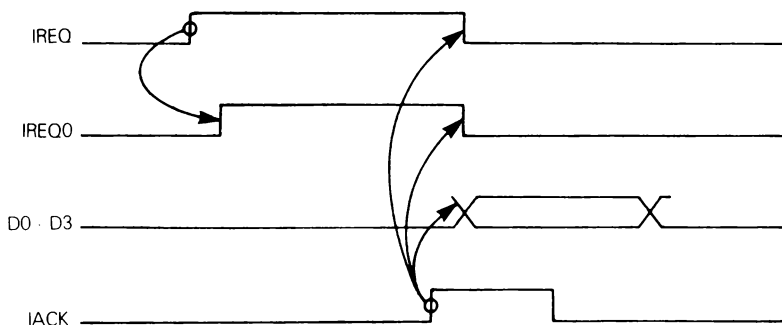
Le priorità d'interruzione non hanno niente a che fare con il fatto che Ⓑ possa essere interrotto o meno dal dispositivo A una volta iniziata l'esecuzione di Ⓒ Il dispositivo ha una priorità d'interruzione minore rispetto al dispositivo C; comunque, una volta che la richiesta d'interruzione del dispositivo C è stata riconosciuta, il dispositivo C annulla la sua richiesta d'interruzione. La richiesta d'interruzione del dispositivo A è ancora presente ed è la richiesta d'interruzione di priorità maggiore. Nell'istante in cui il programma Ⓒ abilita la logica dell'interrupt, esso verrà immediatamente interrotto, e sarà eseguito il programma Ⓐ . Se non volete che il programma Ⓒ sia interrotto, quando scrivete il programma Ⓒ dovete assicurarvi

che esso disabiliti la logica dell'interrupt. L'uso delle istruzioni necessarie per fare ciò è descritto nel Capitolo 7.

PRIORITA' DELL'INTERRUPT E LINEE DI RICHIESTE MULTIPLE

Torniamo ora al dispositivo di priorità dell'interrupt illustrato in Figura 5-13, e spieghiamo come funziona. Come vedete, vi sono 16 linee separate e distinte per mezzo delle quali i dispositivi esterni possono trasmettere i segnali di richiesta di interruzione (IREQ) al dispositivo di priorità dell'interrupt. I segnali terminano ai pin da 10 a 115. 10 ha la priorità massima, mentre 115 ha la priorità minima.

Quando ai pin 10-115 arriva una o più richieste d'interruzione, la logica interna al dispositivo di priorità posiziona IREQ0 a 1 (alto).
In un momento successivo la CPU risponderà posizionando IACK a 1 (alto).

Quando il dispositivo di priorità dell'interrupt riceve lo IACK a 1, posiziona di nuovo IREQ0 a 0, poi mette il numero della linea di richiesta d'interruzione inferiore su D0-D3.
Per esempio, se un IREQ arriva a 15, 0101 è messo in output per mezzo di D0-D3 quando IACK è letto a 1. Se arrivano due segnali IREQ ai pin da 16 a 113, 0110 è messo in output ai pin D0-D3 quando si legge IACK a 1.

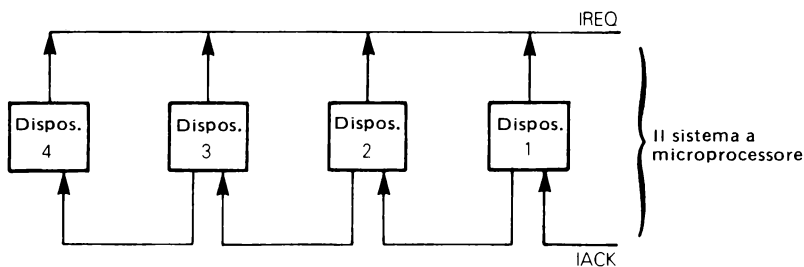
Il Timing è il seguente:

Una volta che la richiesta è stata riconosciuta, i segnali dei dati, di I/O, di READ e di WRITE si usano per trasmettere i dati da e verso la logica esterna. Le linee dati interfacciano il dispositivo di priorità dell'interrupt con il bus del sistema esterno, mentre i pin di I/O interfacciano il dispositivo di priorità con i dispositivi esterni come mostra la Figura 5-14.

I dispositivi esterni possono usare un dispositivo di priorità dell'interrupt per richiedere inizialmente un'interruzione, ma una volta che l'interruzione è stata riconosciuta, il dispositivo esterno può trasferire i dati da e verso il microcomputer tramite le linee dati del bus del sistema esterno. Naturalmente, questo significa che il dispositivo esterno è selezionato per mezzo di indirizzi di memoria, come già descritto in questo Capitolo.

Il punto importante è che qualunque programma eseguito dopo un'interruzione è stato riconosciuto usando la stessa logica con cui era stato riconosciuto il programma eseguito precedentemente. La logica dell'interrupt si applica solo al processo di richiesta e di accettazione di un'interruzione. Se i dispositivi esterni sono collegati al microcomputer tramite le porte di I/O prima dell'interruzione, lo saranno anche dopo che l'interruzione è stata riconosciuta.

PRIORITA' DELL'INTERRUPT E DAISY CHAINING

Se un microcomputer ha una sola linea d'interrupt, o se vi è più di un dispositivo esterno che usa la stessa priorità di richiesta d'interruzione, allora si deve rimpiegare un metodo chiamato "daisy chaining" per determinare le priorità dell'interrupt.

Un certo numero di dispositivi in una daisy chaining saranno tutti collegati alla stessa linea di richiesta d'interruzione IREQ. La linea di riconoscimento dell'interrupt, IACK, comunque, terminerà ad un dispositivo esterno. Questo dispositivo deve avere una logica interna che fa passare il riconoscimento d'interruzione se il dispositivo non sta richiedendo un'interruzione, ma lo blocca in caso contrario. Ogni dispositivo esterno della daisy chain contiene questa stessa logica, ad eccezione dell'ultimo dispositivo, che non può far passare il riconoscimento da nessuna parte. Il "daisy chaining" può essere così configurato:

Il fatto di avere, in un microcomputer, linee di priorità separate, piuttosto che daisy chain, presenta punti di forza e di debolezza. In entrambi i casi il discorso diventa un po' accademico, dato che, come descritto nel Capitolo 6, un microcomputer che viene interrotto da un gran numero di dispositivi esterni, è probabilmente usato nel modo sbagliato. Ciononostante, consideriamo gli aspetti di forza e di debolezza associati alle priorità dell'interrupt separate, e alle daisy chain.

Vi sono situazioni in cui la priorità dell'interrupt separata ha senso, perchè la CPU del microcomputer deve occuparsi di una condizione esterna a spese di tutte le altre.

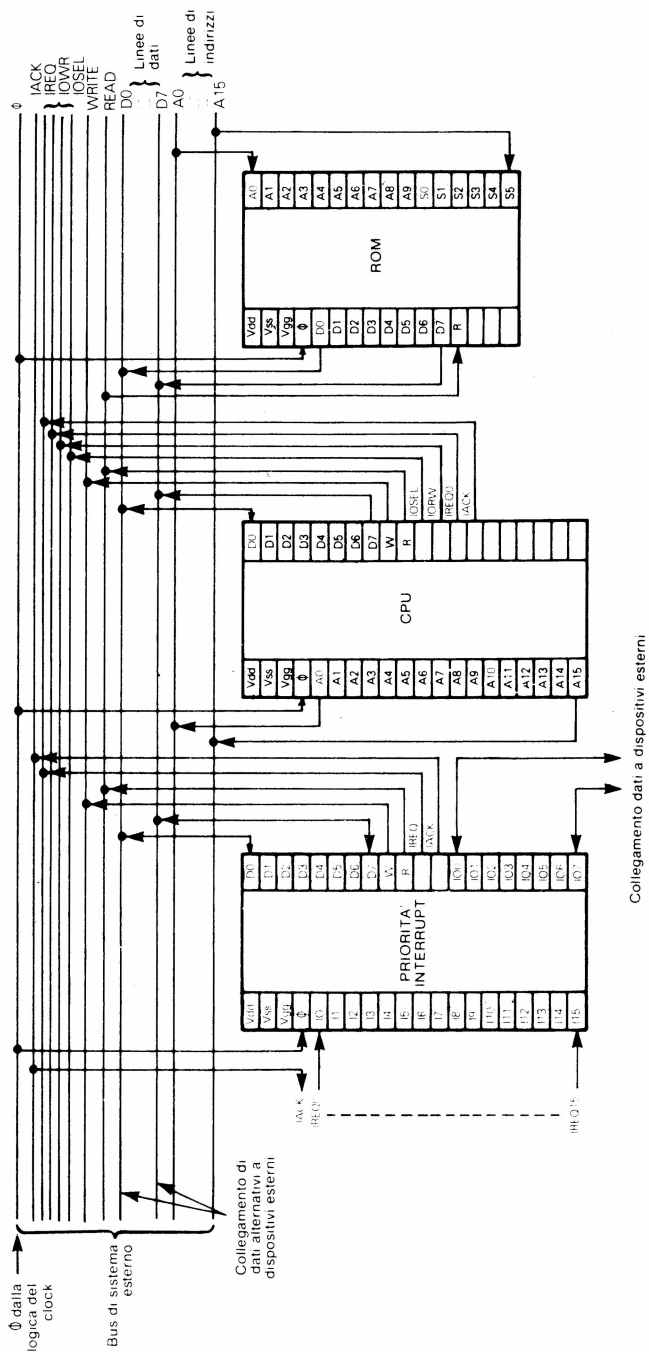



Figura 5-14. Dispositivo di priorità interrupt collegato ad un bus di sistema esterno

Per esempio, la maggior parte dei microcomputer hanno un'interruzione di priorità massima che viene attivata da una caduta di alimentazione. Mentre dapprima questo sembra non avere senso, considerato che cosa succede quando l'alimentazione si interrompe.

INTERRUPT PER CADUTA

Generalmente i microcomputer usano un'alimentazione di +5 e/o +12 volt DC generate dalla normale linea di alimentazione a corrente alternata a 110 volt. La caduta di alimentazione potrebbe essere rivelata quando la tensione scende al di sotto dei 90 volt. Ma possono passare alcuni millesimi di secondi prima che l'alimentazione scenda a tal punto da non poter più fornire +5 e +12 volt per il microcomputer. In questi pochi millisecondi, possono essere eseguite cento o più istruzioni per prepararsi alla caduta di alimentazione in modo da evitare danni gravi; quando l'alimentazione risale di nuovo, il programma interrotto dalla diminuzione dell'alimentazione può ricominciare senza perdere dati.

Abbiamo descritto una situazione in cui ha senso una linea di priorità dell'interrupt separata. **Consideriamo ora i limiti del daisy chaining.**

Il daisy chaining riesce a trattare un certo numero di dispositivi, che richiedono tutti il servizio d'interruzione, finchè questo numero non diventa troppo grande. Consideriamo quanto poco sarebbe servita la centesima doccia se il microcomputer deve prima rispondere alle necessità delle 99 docce precedenti nella daisy chain. E' concepibile che il microcomputer sarà così occupato a dedicarsi ai dispositivi situati all'inizio della daisy chain che i dispositivi della parte finale riceverebbero pochissima o nessuna attenzione. L'occupante della camera 100 verrà ustionato o ghiacciato.

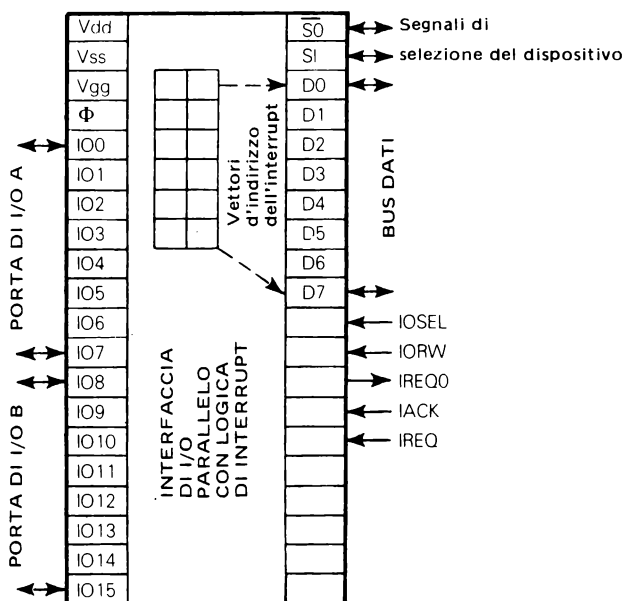
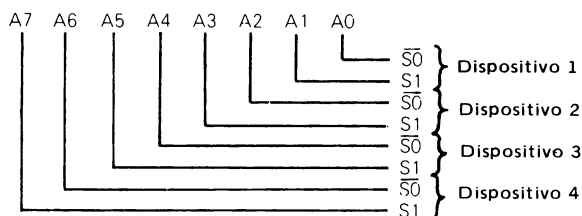
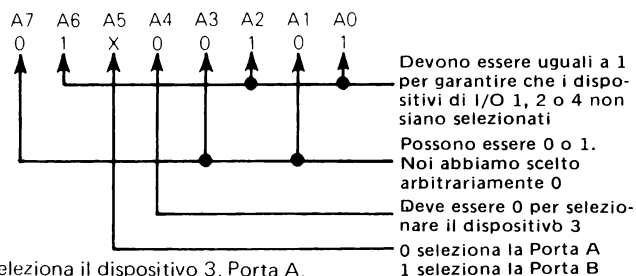
Un altro problema del daisy chaining è che esso richiede intelligenza in ogni dispositivo della daisy chain. Ancora una volta, ci stiamo occupando dell'economia del microcomputer. I dispositivi esterni in una daisy chain devono identificare se stessi, altrimenti il microcomputer non ha modo di sapere a che punto della daisy chain è andato il segnale di riconoscimento dell'interrupt IACK lanciato prima di venire bloccato. Così ci troviamo di nuovo a richiedere che i dispositivi esterni della daisy chain contengano logica sufficiente per bloccare il segnale di riconoscimento quando viene richiesta un'interruzione, e poi per trasmettere un codice di identificazione dispositivo al microcomputer. Certamente questa logica potrebbe essere implementata per pochi dollari, ma ricordate del resto che un microcomputer non ne costa molti.

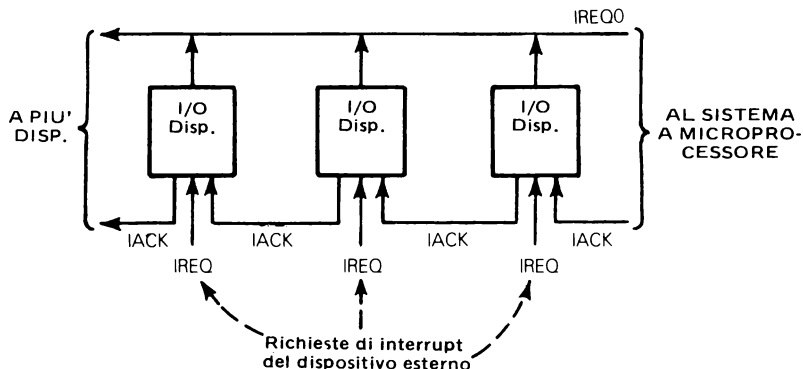
IL "DAISY CHAINING" CON DISPOSITIVI DI INTERFACCIA DELL'I/O

Al fine di eliminare il costo della logica esterna richiesta per implementare una daisy chain, alcuni costruttori di microcomputer forniscono questa logica su dispositivi di supporto. Consideriamo la logica della daisy chain su di un dispositivo di interfaccia dell'I/O; nella pagina che segue è rappresentata una possibile configurazione di uno di questi dispositivi.

Allo scopo di trovare i pin in più per i segnali necessari, abbiamo sostituito 8 pin di selezione dispositivi (A0-A7) nella Figura 5-12 con due pin, $\overline{\text{S0}}$ e S1. Perchè il dispositivo venga selezionato, deve essere inserito un segnale 0 a $\overline{\text{S0}}$. In input o simultaneo a S1 selezionerà la porta A. Un input 1 simultaneo a S1 selezionerà la porta B.

LOGICA DI SELEZIONE DISPOSITIVO

Di solito potete scegliere due linee d'indirizzamento e legarle insieme direttamente a $\overline{\text{S0}}$ e S1, selezionando così i dispositivi di I/O in parallelo usando molta della logica di selezione del dispositivo esterno.

Ecco come potreste usare le 8 linee d'indirizzamento illustrate in Figura 5-12 per selezionare quattro dispositivi di I/O in parallelo:

Quali indirizzi dovrebbero essere assegnati alle porte di I/O A e B del dispositivo 3? Potete elaborare gli indirizzi nel modo seguente:

Così 45_{16} seleziona il dispositivo 3, Porta A.
65_{16} seleziona il dispositivo 3, Porta B.

Questa logica di selezione dispositivo non ha niente a che fare con gli interrupt — che sono l'argomento che stiamo attualmente discutendo. Più dispositivi d'interfaccia dell'I/O in parallelo con la logica degli interrupt verrebbero usati in questo modo:

Ecco cosa succede:

1) Quando uno o più dispositivi esterni richiedono un'interruzione, posizionano il loro IREQ "vero".
2) Quando uno o più dispositivi di I/O in parallelo ricevono un IREQ "vero", lo inoltrano alla CPU tramite la linea comune IREQ0.
3) La CPU riceve un input IREQ0 "vero". La logica della CPU ora sa che un dispositivo — non sa quale — sta richiedendo un'interruzione.
4) Quando la logica della CPU permette che l'interrupt venga riconosciuto, mette in uscita IACK "vero".
5) Il primo dispositivo di I/O in parallelo nella daisy chain riceve IACK "vero"; esso blocca il segnale IACK, a prescindere dal fatto che esso sia stato o meno selezionato per mezzo delle linee di selezione S0 e S1. Se il primo dispositivo di I/O in parallelo nella daisy chain ha un ingresso IREQ "falso", esso inoltra il segnale IACK al dispositivo di I/O 2. Il primo dispositivo di I/O con ingresso IREQ "vero" blocca lo IACK.
6) Il dispositivo di I/O che blocca lo IACK ora mette in output il suo vettore d'indirizzo dell'interruzione sul bus; ciò permette al dispositivo di I/O di essere identificato. Notate che la logica di selezione associata con $\overline{S0}$ e S1 non è ancora coinvolta. La logica di riconoscimento/richiesta d'interruzione ha la sua logica di selezione propria — è parte della logica che blocca lo IACK e lo fa passare.
7) Ora il programma di servizio dell'interrupt ha termine. Se devono essere inseriti o messi in output dei dati, si usano $\overline{S0}$ e S1 per selezionare il dispositivo di I/O appropriato, esattamente come farebbe qualunque altro programma.

DISPOSITIVI CON PIU' FUNZIONI

Il dispositivo I/O in parallelo con logica dell'interrupt introduce un concetto molto importante ed è il fatto di mettere più di una funzione su di un solo chip. La logica di I/O in parallelo e la logica dell'interrupt non hanno niente in comune. Capi-

ta che siano sullo stesso chip, perciò condividono i pin del bus dati da D0 a D7.

La logica esterna può richiedere un'interruzione per mezzo di un dispositivo di I/O; poi dopo che l'interrupt è stato riconosciuto, la logica esterna può trasmettere o ricevere dati per mezzo delle porte di I/O, dello stesso dispositivo di I/O, di un altro dispositivo di I/O, o di nessun dispositivo.

In seguito ad un interrupt, non vi è nessuna ragione per cui dobbiate trasferire i dati per mezzo delle porte di I/O del dispositivo di I/O che ha bloccato IACK.

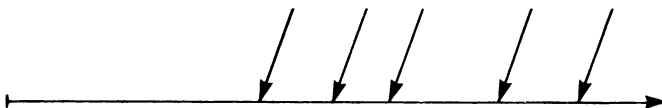
Capita che la logica di I/O e la logica dell'interruzione si trovino nello stesso chip; esse non hanno nessun altro punto in comune.

ACCESSO DIRETTO IN MEMORIA (DIRECT MEMORY ACCESS – DMA)

Il microcomputer che controlla la temperatura della doccia spenderà moltissimo del suo tempo semplicemente a ricevere i dati dal lettore di temperatura e a memorizzarli in un buffer RAM.
Abbiamo spiegato adesso che gli interrupt possono essere usati per eseguire il programma RECORD in qualunque momento il lettore di temperatura sia pronto per trasmettere i dati. Riguardiamo un'altra volta, più attentamente, questo argomento.

Ricordate che il lettore di temperatura può trasmettere circa due letture di temperature al secondo. Per il microcomputer, ciò equivale a ricevere una lettura di temperatura una volta circa ogni quarto di milione di esecuzioni di istruzioni.

EVENTI ASINCRONI

Un lettore di temperature economico non trasmetterà esattamente due letture della temperatura ogni secondo, e quindi vi potrebbero essere delle considerevoli variazioni del tempo che intercorre tra ogni trasmissione di temperatura. Come risultato, non possiamo prevedere, con nessun grado di precisione, il ritardo di tempo fra le trasmissioni di dati consecutive dal lettore della temperatura al microcomputer. Per questo motivo le trasmissioni di dati dal lettore di temperatura al microcomputer costituiscono eventi asincroni.

Poichè le trasmissioni di dati dal lettore della temperatura al microcomputer sono eventi imprevedibili (o asincroni), il programma RECORD deve essere eseguito nel momento in cui il lettore di temperatura trasmette un numero di dati. Il programma RECORD contiene la sequenza di istruzioni che sposterà i dati da una porta di I/O ad un byte nel buffer di memoria RAM; questa sequenza di istruzioni non può far parte del programma ADJUST, dato che la logica di tale programma non può rilevare l'arrivo dei dati da un lettore di temperature. Qualunque schema che esegua la se-

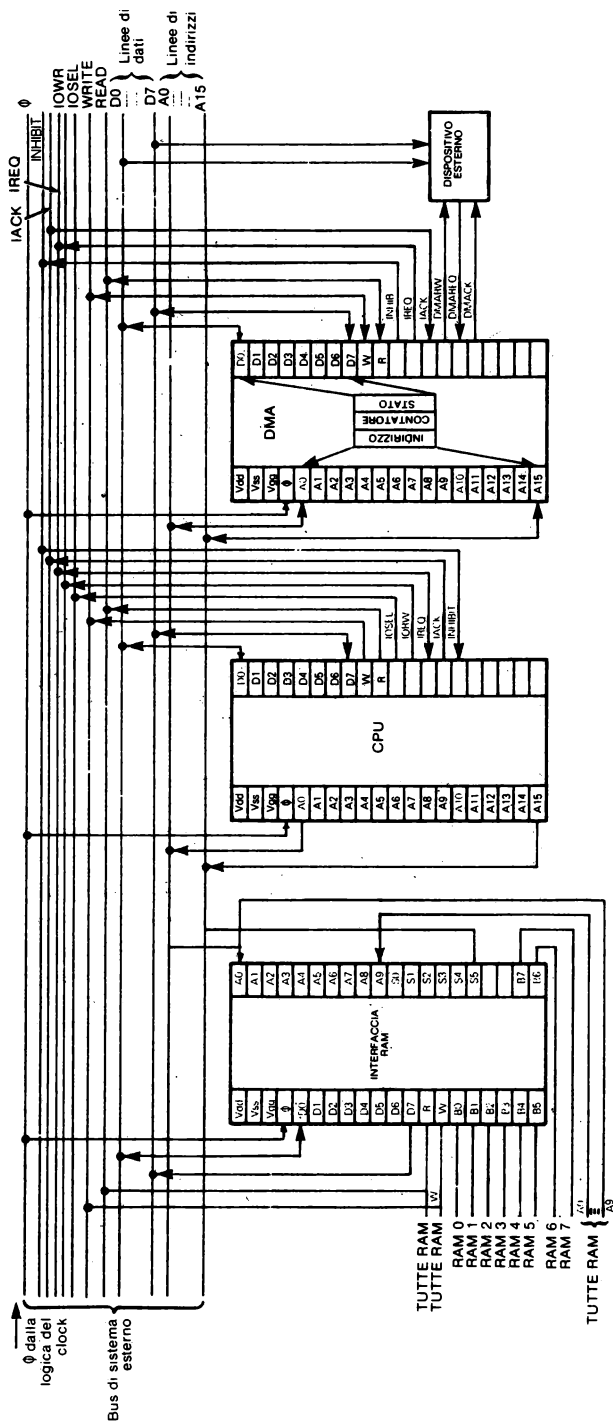
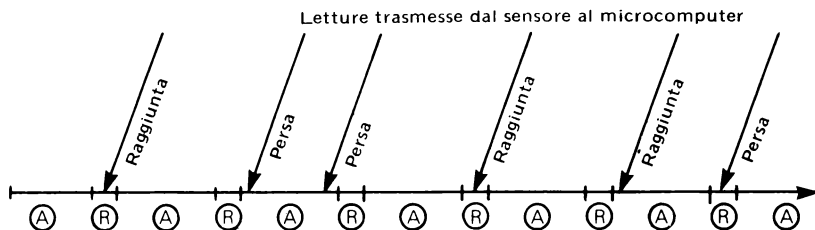
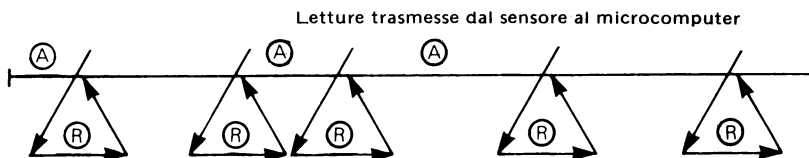



Figura 5-15. Ciclo di accesso diretto alla memoria

quenza delle istruzioni di RECORD a intervalli di tempo fissati, è costretto a perdere un certo numero delle trasmissioni di dati dal lettore di temperatura. Ecco un esempio:

Ⓐ rappresenta la normale esecuzione del programma ADJUST. Ⓡ rappresenta l'esecuzione fissa, periodica della sequenza di istruzioni che registra i dati trasmessi dal lettore di temperatura al microcomputer. Se i dati trasmessi non raggiungono il microcomputer durante Ⓡ , possono andare perduti.

L'unico modo sicuro di ricevere tutti i dati trasmessi dal lettore di temperature è che quest'ultimo richieda un'interruzione quando è pronto per trasmettere un dato. In risposta alla richiesta d'interruzione del lettore di temperatura, il microcomputer eseguirà la sequenza di istruzioni di lettura dati, caratterizzata nella figura precedente da Ⓡ La figura deve essere ora modificata in questo modo:

Ogni volta che il lettore di temperatura trasmette i dati al microcomputer, lo notifica alla CPU richiedendo un'interruzione. In risposta alla richiesta d'interruzione, la logica del programma sospende l'esecuzione del programma ADJUST (Ⓐ) per eseguire il programma RECORD (Ⓡ). Il programma RECORD legge i dati in input dal lettore di temperatura, poi il programma ADJUST continua l'esecuzione dal punto in cui era stata sospesa.

Anche questo metodo di registrare i dati trasmessi dal lettore di temperatura non è molto efficiente. **Ecco che cosa succede quando la CPU accetta un interrupt ed esegue RECORD:**

1) La CPU sta eseguendo il programma ADJUST (Ⓐ) come nell'illustrazione precedente. Quando la CPU sente una richiesta d'interruzione, esegue delle istruzioni che salvano il contenuto dei registri della CPU e lo stato; poi esegue un'istruzione per mandare il riconoscimento dell'interrupt.
2) Il programma RECORD (Ⓡ) nell'illustrazione precedente viene eseguito. Questo programma contiene istruzioni che caricano un indirizzo di memoria nel data counter, leggono i dati da una porta di I/O nell'accumulatore, poi mettono in output i dati dell'accumulatore nella parola di memoria indirizzata dal Data Counter.
3) Il passo 3 è invertito. Il contenuto dei registri e lo stato precedentemente salvati vengono ripristinati e il programma ADJUST prosegue l'esecuzione.

In tutte le istruzioni che vengono eseguite per implementare i tre passi suddetti, l'unico cambiamento, ogni volta che il lettore di temperatura trasmette una lettura e Ⓡ è rieseguito, è il contenuto del Data Counter del passo 2.

Il contenuto del Data-Counter sarà maggiore di uno rispetto alla volta precedente, e minore di uno rispetto alla volta successiva. Sono necessari cinquanta microsecondi o più per elaborare ripetitivamente una sequenza di eventi altrimenti prevedibili e non significativi. Prima di poter decidere se questo è un problema serio o senza conseguenze, dobbiamo porci due domande:

1) **Le operazioni di questo tipo sono molto comuni, o questa è una situazione isolata e speciale?**
   La risposta è che questa è una delle operazioni più comuni eseguite dal microcomputer. Infatti, non solo il computer impiegherà molto tempo a leggere i dati da un dispositivo esterno, ma impiegherà quasi lo stesso tempo a trasmettere con una routine i dati dai buffer della RAM ai dispositivi esterni.

2) **Se il microcomputer non impiegasse 50 microsecondi ogni volta che prende in input dati da un dispositivo esterno (o li mette in output su di un dispositivo esterno) che cosa farebbe ancora nel tempo rimanente?**
   In molte applicazioni semplici la risposta è niente, nel qual caso il tempo sprecato è irrilevante. Ma chiaramente, quando l'applicazione di un microcomputer incomincia a diventare complessa, gli sprechi di tempo incominciano a diventare seri. Se occorrono 50 microsecondi per leggere un certo numero di dati da un dispositivo esterno, ed altri 50 microsecondi per trasmettere un certo numero di dati allo stesso dispositivo esterno, allora un microcomputer potrebbe eseguire cento trasferimenti di dati ogni secondo — ma non vi sarebbe più tempo per fare niente altro.

Dobbiamo perciò concludere che vi sarà un numero significativo di applicazioni in cui il tempo sprecato nell'elaborare gli interrupt sarà intollerabile.

ACCESSO DIRETTO IN MEMORIA CON SOTTRAZIONE DI CICLI

L'accesso diretto in memoria (DMA) fornisce una soluzione. Creeremo un nuovo dispositivo per il nostro microcomputer e su questo dispositivo metteremo una piccola quantità di logica di tipo CPU, dedicata al solo compito di spostare i dati fra le porte di I/O (o le linee dati del bus del sistema esterno) e la memoria.

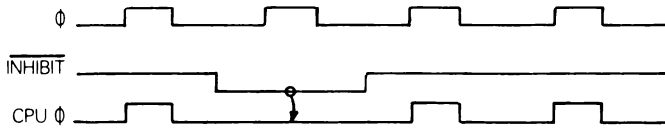
Il dispositivo DMA svolgerà il suo compito eliminando o superando la logica della CPU mentre crea le sequenze di segnali che abilitano il trasferimento di dati appropriato.

Una volta data l'architettura del microcomputer, come quella che è stata descritta in questo capitolo, il compito di progettare un dispositivo DMA è del tutto definito.

CONTROLLO DELL'INIBIZIONE

Eliminare temporaneamente la logica della CPU è facile, dato che la CPU è guidata da un segnale di clock esterno. Se possiamo fermare il segnale di clock, possiamo fermare la CPU. Perciò come mostra la Figura 5-15, aggiungeremo una linea di controllo INHIBIT al nostro bus del sistema esterno, e un pin di controllo INHIBIT al dispositivo della CPU.

Il segnale $\overline{\text{INHIBIT}}$ (o di inibizione) sarà normalmente a 1, in modo che un semplice gate AND interno al chip della CPU combinerà il segnale di clock con il segnale $\overline{\text{INHIBIT}}$, per generare il segnale di clock interno che guida la logica della CPU.

Tutto quello che il dispositivo DMA deve fare è di mettere il segnale di $\overline{\text{INHIBIT}}$ a 0, è così eliminare i segnali di clock del timing all'interno della CPU:

I dispositivi di memoria e i dispositivi di interfaccia in parallelo non hanno modo di sapere da dove vengono i segnali di controllo che guidano la loro logica. Perciò, il dispositivo DMA può assumere il controllo del bus del sistema mentre la logica della CPU è soppressa.

Vediamo più dettagliatamente la Figura 5-15. Il dispositivo DMA, come vedete, contiene i tre registri seguenti:

1) **Un registro indirizzi** che contiene l'indirizzo della successiva parola di memoria cui bisogna accedere, per un'operazione di lettura o di scrittura.

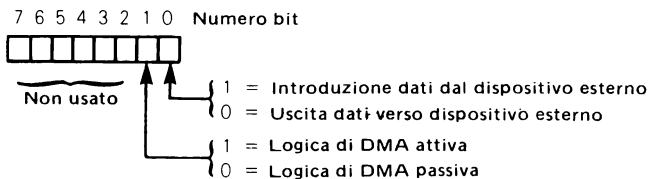
2) **Un contatore**, che inizialmente contiene la lunghezza del buffer dati che deve essere riempito durante un'operazione di lettura, o dal quale si devono leggere i dati durante un'operazione di scrittura.

3) **Un registro di stato**, che identifica la direzione del flusso dati, se la logica del DMA è attualmente attiva o inattiva, e molte altre scelte del DMA. Ci occuperemo di alcune di queste scelte più avanti.

Alla CPU, i registri del DMA possono apparire come porte di I/O o parole di memoria indirizzabili. Inizialmente le istruzioni di I/O programmate saranno usate per posizionare i valori nei registri indirizzi, nel contatore e nel registro di stato. Ecco un esempio.

Valore	Registro
0080	Registro indirizzi
007F	Registro contatore
03	Registro di stato

Questo semplice esempio specifica che un buffer dati lungo $7F_{16}$ byte e con inizio a 0080_{16}, deve essere riempito con i dati che provengono da un dispositivo esterno, usando l'accesso diretto in memoria.

Il registro di stato è per il momento limitato a questo semplice formato:

Mettendo il valore 03_{16} nel registro di stato, viene specificato l'input dei dati da un dispositivo esterno, e attivata la logica del DMA.

INIZIALIZZAZIONE DEL DMA

La CPU deve eseguire un programma per inizializzare un'operazione DMA. Il programma dovrà caricare i dati in maniera appropriata nei registri del dispositivo DMA. Non vi è un altro modo di inserire i dati nei registri indirizzi, contatore e nel registro di stato del dispositivo DMA.

Allo scopo di inizializzare un'operazione DMA, la CPU dovrebbe eseguire un programma per realizzare questi passi:

1) Trasmettere la parte meno significativa dell'indirizzo d'inizio al dispositivo DMA.
2) Trasmettere la parte più significativa dell'indirizzo d'inizio al dispositivo DMA.
3) Trasmettere la lunghezza del buffer al dispositivo DMA.
4) Trasmettere un codice di controllo (03_{16}, come già visto) al registro di stato del dispositivo DMA. Il codice di controllo identifica la direzione del trasferimento dati e deve attivare il dispositivo DMA.

Da notare che il dispositivo DMA ha linee di controllo di READ e di WRITE.
La linea di controllo WRITE verrà usata dal dispositivo DMA per scrivere i dati nella RAM.

La linea di controllo di READ verrà usata dal dispositivo DMA per mettere in output i dati della RAM o della ROM.

LA PRESA AL VOLO DEL DMA

La CPU può in qualunque momento, leggere il contenuto dei registri del dispositivo DMA. Questo permette al programma di controllare quanto sia avanzata un'operazione DMA mediante la lettura del contenuto del registro indirizzi e/o del registro contatore. **Un programma che adatta la sua logica al livello in cui si trova in quel momento l'esecuzione di un'operazione DMA, si dice che prende al volo l'accesso diretto in memoria.**

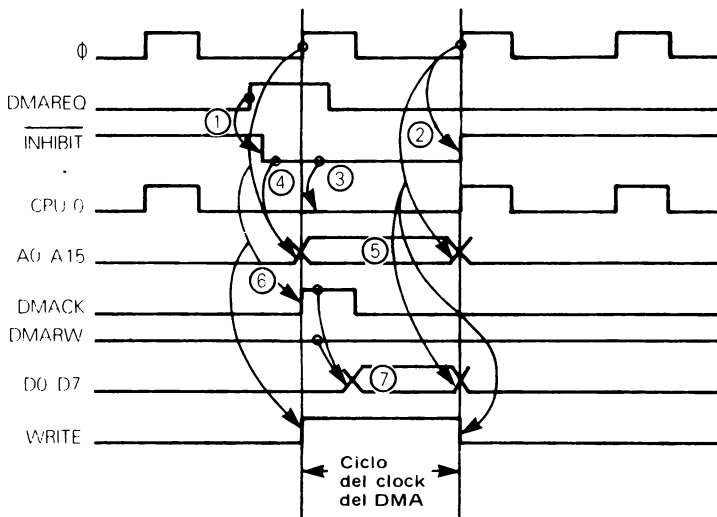
INTERRUZIONE DEL DMA

Naturalmente, un programma in corso può interrompere un'operazione DMA in qualunque momento semplicemente scrivendo nuovi dati nel registro di stato del dispositivo DMA. Come già descritto, posizionando il bit 1 del registro di stato a 0 si arresta immediatamente qualunque trasferimento dati DMA in corso.

ESECUZIONE DEL DMA

Che cosa succede dopo che è iniziata un'operazione DMA? Notate che i dispositivi esterni sono collegati direttamente alle linee dati del bus dati esterno. Inoltre, i dispositivi esterni hanno un segnale di richiesta di accesso diretto in memoria (DMAREQ), che, quando è a 1, ha influenza sul dispositivo DMA. Se i dati vengono letti da un dispositivo esterno, si verifica la sequenza di segnali come raffigurato nella pagina che segue.

Tutta l'operazione DMA avviene in un ciclo di clock.
Gli eventi si svolgono in questo modo:

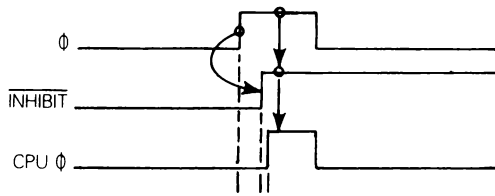
1) Appena il dispositivo DMA sente un impulso a 1 sulla linea DMAREQ, mette immediatamente a 0 il segnale di controllo $\overline{\text{INHIBIT}}$ (① nella fig.). Il controllo $\overline{\text{INHIBIT}}$ viene mantenuto a 0 finchè non arriva il prossimo fronte di salita dell'impulso Φ (② nella fig.).

LO SPIAZZAMENTO DEI BUS

2) la CPU sospenderà le operazioni per un ciclo di clock perchè $\overline{\text{INHIBIT}}$ mantiene a 0 la linea Φ della CPU (③ nella fig.). La CPU applicherà anche una forte resistenza alle connessioni dei bus d'indirizzo e dati, scollegandosi effettivamente da questi bus. Questo viene chiamato spiazzamento dei bus.

3) La combinazione di $\overline{\text{INHIBIT}}$ a 0 e di Φ che passa a 1 (④ nella fig.) fa sì che il dispositivo DMA metta in output il contenuto del suo registro indirizzi sulle linee d'indirizzamento del bus del sistema esterno (⑤ nella fig.). Il dispositivo DMA riconosce anche la richiesta DMA mettendo a 1 la linea di controllo della risposta DMACK (⑥ nella fig.).

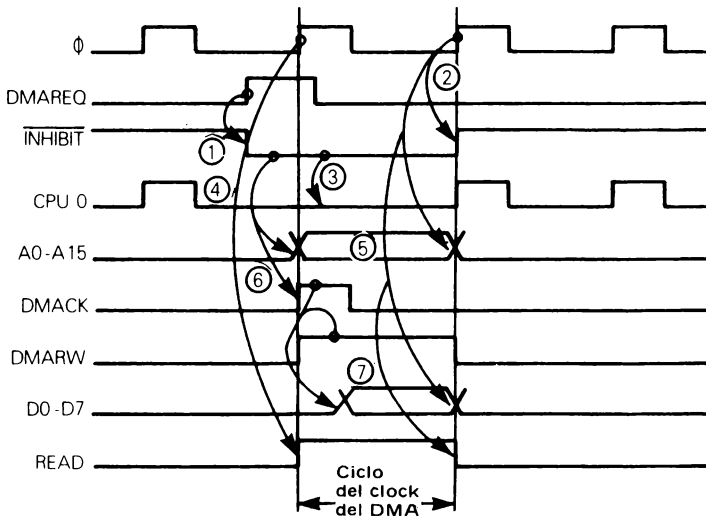
4) Il bit 0 del registro di stato DMA determina se la linea di controllo DMARW sarà a 1 o a 0. Nome già illustrato, è a 1, indicando alla logica esterna che deve trasmettere i dati alle linee dati del bus del sistema esterno. La combinazione di DMACK alto e di DMARW basso fa sì che il dispositivo esterno metta i suoi dati sulle linee dati del bus del sistema esterno (⑦ nella fig.).

5) Il bit 0 del registro di stato DMA fa sì anche che il dispositivo DMA metta in uscita un 1 sulla linea di controllo WRITE. Tutti i dispositivi d'interfaccia della RAM decodificheranno l'indirizzo sulle linee d'indirizzo e una si troverà selezionata; sentendo il segnale di controllo WRITE a 1 il dispositivo d'interfaccia della RAM selezionata prenderà il dato presente sulle linee dati del bus del sistema esterno e lo scriverà nella parola di memoria indirizzata. La logica dell'interfaccia della ROM non sa e non le interessa sapere dove vengono originati i dati, l'indirizzo e i segnali di controllo. Essa risponde semplicemente a qualunque situazione che attivi la sua logica interna.

6) Come già visto, il secondo fronte di salita di Φ interrompe le operazioni. In realtà si dovrebbero usare degli altri schemi, dato che quello illustrato, sebbene molto semplice, provocherebbe ripercussioni sul fronte principale della CPU Φ. Il Φ della CPU diventa 1 solo perchè $\overline{\text{INHIBIT}}$ sale perchè Φ è salito per la seconda volta. Chiaramente, l'ascesa di Φ per la seconda volta non può coincidere esattamente con l'ascesa del Φ della CPU, perchè nel frattempo $\overline{\text{INHIBIT}}$ è passata a 1 e deve stabilizzarsi dopo l'ascesa.

Così, come abbiamo visto, il diagramma di timing della lettura DMA è concettualmente accurato ma non realistico.

Non appena il dispositivo DMA ha messo in output il contenuto del suo registro indirizzi sul bus dati esterno, incrementa il contenuto del registro, in modo da indirizzare la parola successiva del buffer dati della RAM. Simultaneamente il chip DMA decrementerà il contenuto del suo registro contatore:

TIMING DELLA SCRITTURA COL DMA

Ecco la sequenza di segnali per un'operazione di scrittura DMA:

Notate che l'operazione di scrittura col DMA è diversa dall'operazione di lettura col DMA solo per il fatto che il dispositivo DMA posiziona il segnale di controllo READ a 1 quando mette il contenuto del suo registro indirizzi sulle linee d'indirizzo del bus dati esterno. Questo fa sì che il modulo di memoria che è selezionato dallo indirizzo di memoria metta il contenuto della parola di memoria indirizzata sulle linee del bus dati esterno. DMARW, posizionato a 1 dal chip DMA, fa sì che il dispositivo esterno legga il contenuto del bus dati.

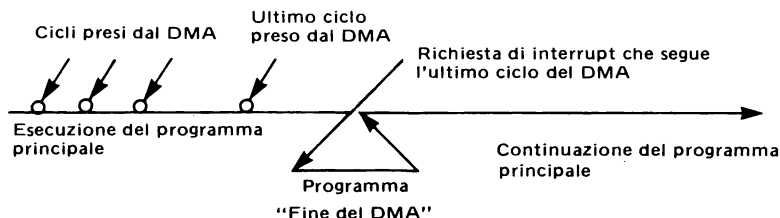
Ancora una volta il dispositivo DMA decrementerà il suo registro contatore e incrementerà il suo registro indirizzi, in modo da essere pronto per l'operazione DMA successiva.

Quando il registro contatore è decrementato a zero, può succedere una di queste due cose:

FINE DEL DMA

1) **Il dispositivo DMA può segnalare il fatto che l'operazione DMA è finita mandando una richiesta d'interruzione alla CPU.** Questo interrupt dovrebbe essere trattato secondo la logica di elaborazione dell'interrupt (qualunque esso sia) che la CPU sta usando.

2) **Il dispositivo DMA può semplicemente ricominciare di nuovo l'intero processo, conservando il valore originale del registro contatore e del registro indirizzi,** in modo da poter ricaricare questi valori originali e permettere alle operazioni di procedere senza fermarsi, o finchè vengono fermate dalla CPU.

Le due scelte disponibili quando il dispositivo DMA decrementa fino a zero sono così illustrate:
Prima la "fine del DMA e interrupt"

Consideriamo quindi un dispositivo DMA con registri di memorizzazione aggiuntivi per il conteggio dell'indirizzo iniziale e della lunghezza del buffer:

Quando il valore corrente del contatore diventa 0, il valore iniziale del contatore è caricato nel valore corrente del contatore, e l'indirizzo iniziale è caricato in indirizzo corrente; lo stato resta invariato (a meno che non venga modificato dalla CPU), e l'operazione DMA ricomincia, accedendo allo stesso buffer, dallo stesso indirizzo di inizio.

DMA CON DISPOSITIVI ESTERNI MULTIPLI

Un dispositivo DMA può controllare le operazioni DMA per molti dispositivi esterni. Dato che un dispositivo DMA non trasferisce nessun dato, non ha bisogno di nessuna porta di I/O attraverso la quale i dispositivi esterni comunichino con il microcomputer. I dispositivi esterni sono collegati direttamente alle linee dati del bus del sistema esterno. Questa disposizione è molto conveniente, dato che permette al dispositivo DMA di controllare l'accesso DMA per un certo numero di dispositivi esterni.

Si potrebbero escogitare molti schemi che permettono ad un dispositivo DMA di controllare l'accesso DMA per più di un dispositivo esterno: la Figura 5-16 illustra una possibilità.

Il dispositivo DMA illustrato nella figura suddetta controlla l'accesso DMA per cinque dispositivi esterni. Ogni dispositivo esterno ha la sua linea di richiesta DMA (DMAREQ1

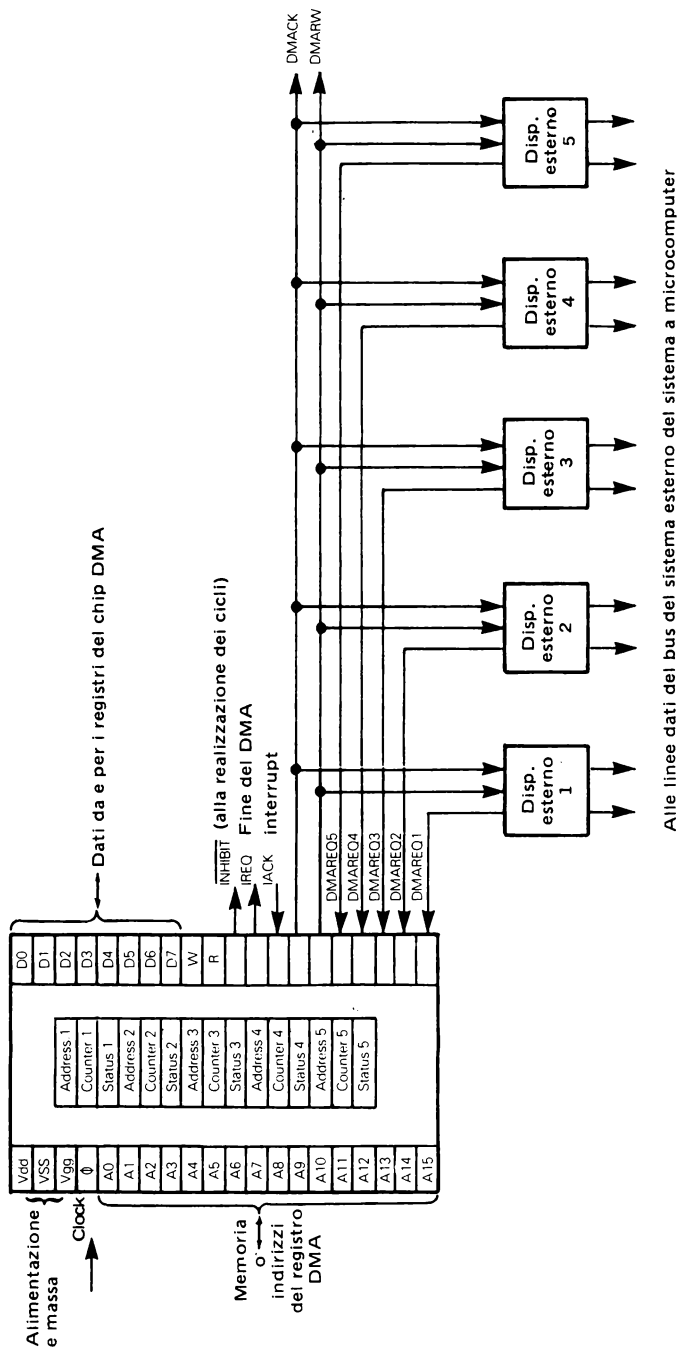
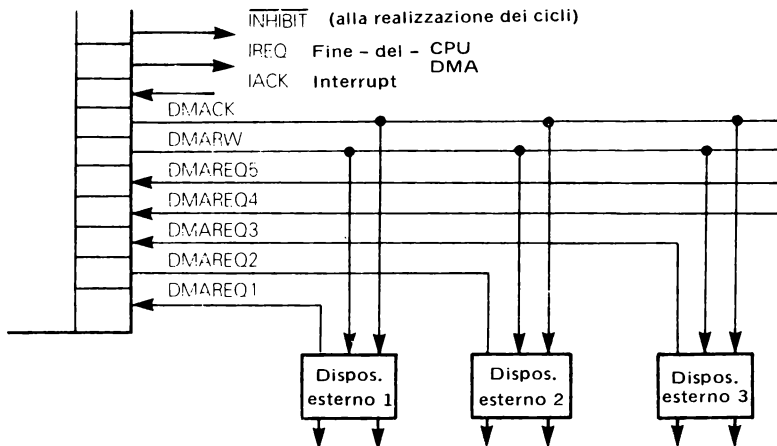



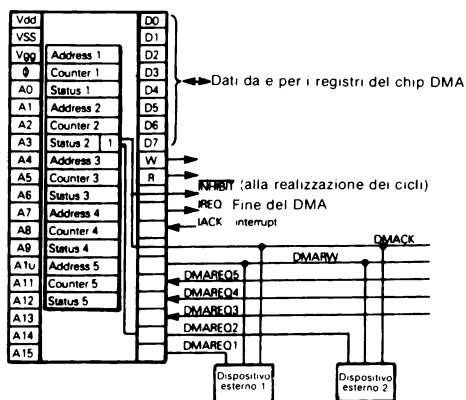
Figura 5-16. Dispositivo DMA che controlla le operazioni di cinque dispositiv esterni

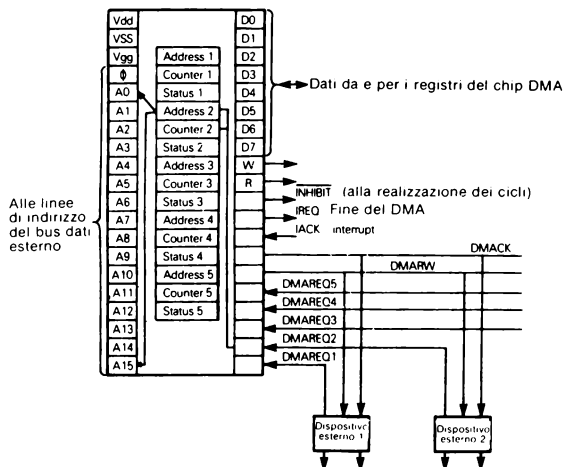
fino a DMAREQ5). Le linee comuni di riconoscimento DMA (DMACK) e le linee di controllo di lettura-scrittura (DMARW) vengono usate da tutti i dispositivi. Vi sono cinque set di registri all'interno del dispositivo DMA, un set per ogni dispositivo esterno. Ogni dispositivo esterno, quando è pronto per trasmettere o ricevere dati, indica questo fatto posizionando la sua linea di richiesta DMA a 1. La logica del dispositivo DMA accede ai tre registri dati interessati, sulla richiesta DMA ricevuta. Per esempio, DMAREQ3 a 1 identifica l'indirizzo 3, il contatore 3 e lo stato 3 ma i registri che contengono i dati che devono essere usati in questo caso.

Quando il dispositivo DMA per più dispositivi illustrato in Figura 5-16 sottrae un ciclo e trasferisce una parola dati per mezzo del DMA, la sequenza di segnali è identica a quella che abbiamo già descritto per un solo dispositivo esterno. Comunque, i dispositivi esterni della Figura 5-16 devono avere la loro logica di selezione. In altre parole, un dispositivo che raggiunge la sua linea di richiesta DMA deve essere il solo dispositivo a rispondere a DMACK, DMARW e al bus dati esterno; nessun altro dispositivo attaccato al dispositivo DMA a più dispositivi deve rispondere a queste linee. E' responsabilità del dispositivo esterno, non del dispositivo DMA a più dispositivi, assicurarsi che un solo dispositivo esterno si consideri in ogni momento selezionato.

Finchè un solo dispositivo esterno si considera selezionato per DMA non vi è possibilità di confusione nel trasferimento dei dati DMA. I moduli di memoria rispondono solamente all'indirizzo e ai segnali di controllo messi in uscita dal dispositivo DMA. I moduli di memoria non sanno nè si interessano di sapere dove ha avuto origine questa informazione. Solo il dispositivo esterno selezionato è attivo all'altro estremo del bus del sistema esterno, così i due confini del trasferimento dati sono chiaramente definiti.

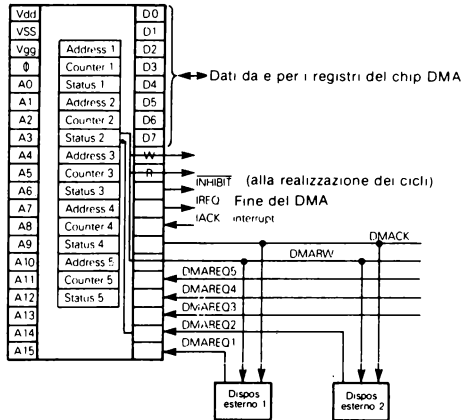
Consideriamo un esempio nei particolari.
Il dispositivo esterno 2 è pronto per un altro accesso ai dati, così mette DMAREQ a 1:

La prima cosa che la logica del dispositivo DMA farà è controllare il registro di stato associato con DMAREQ2, in questo caso il registro di stato 2. Se il bit di abilitazione (bit 0) è 0, la logica del dispositivo DMA ignorerà la richiesta DMA. Dato che DMAREQ2 è un segnale impulsivo, tornerà a 0. Se il bit di abilitazione 7 1, la logica

del dispositivo DMA riconoscerà l'interrupt su DMACK e sottrarrà un ciclo della CPU mettendo $\overline{INHIBIT}$ a 0:

Per l'operazione DMA in arrivo, solo il dispositivo esterno 2 è attivato sul confine esterno del trasferimento dei dati.

DMAREQ2 fa sì che il contenuto del registro indirizzi 2 venga messo in output ai pin d'indirizzo.

La logica del dispositivo DMA decrementerà il contenuto del contatore 2 e incrementerà il contenuto dell'indirizzo 2.

Dato che DMAREQ2 identifica l'indirizzo 2 come il registro DMA contenente l'indirizzo di memoria richiesto, non può nascere nessuna confusione dal fatto che sono presenti altri quattro indirizzi, in altri quattro registri.

DMAREQ2 identifica anche lo stato 2 come il registro di stato che controlla le opera-

zioni in corso. Le linee di controllo W, B e DMARW sono posizionate basandosi sui contenuti dello stato 2:

Ora avviene un trasferimento dati fra la parola di memoria indirizzata dall'indirizzo 2 e il dispositivo esterno 2. Le sequenze di segnali associate con il trasferimento dati sono esattamente come descritte per il chip DMA con un singolo dispositivo esterno.

DMA SIMULTANEO

Osservate che l'accesso diretto in memoria con sottrazione di cicli non implica niente altro che la riproduzione di una quantità limitata della logica della CPU su di un dispositivo DMA. **Un dispositivo DMA può essere paragonato ad una CPU che è capace di eseguire solo due istruzioni:**

1) Trasferire dati da un dispositivo esterno alla memoria.

2) Trasferire dati dalla memoria ad un dispositivo esterno.

A causa del numero molto limitato di operazioni che il dispositivo DMA può eseguire, quasi tutte le sequenze che sprecano tempo associate con le operazioni della CPU (per esempio il prelevamento di un'istruzione) possono essere eliminate.

Ma possiamo spingere la logica del DMA un passo più avanti. Duplicando parte del bus del sistema esterno, possiamo eliminare il bisogno di sottrarre cicli alla CPU. Riguardiamo il diagramma di timing per un'operazione di lettura in memoria:

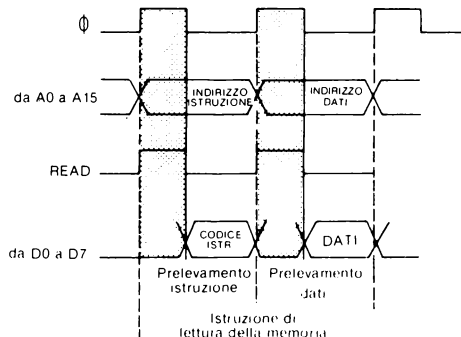
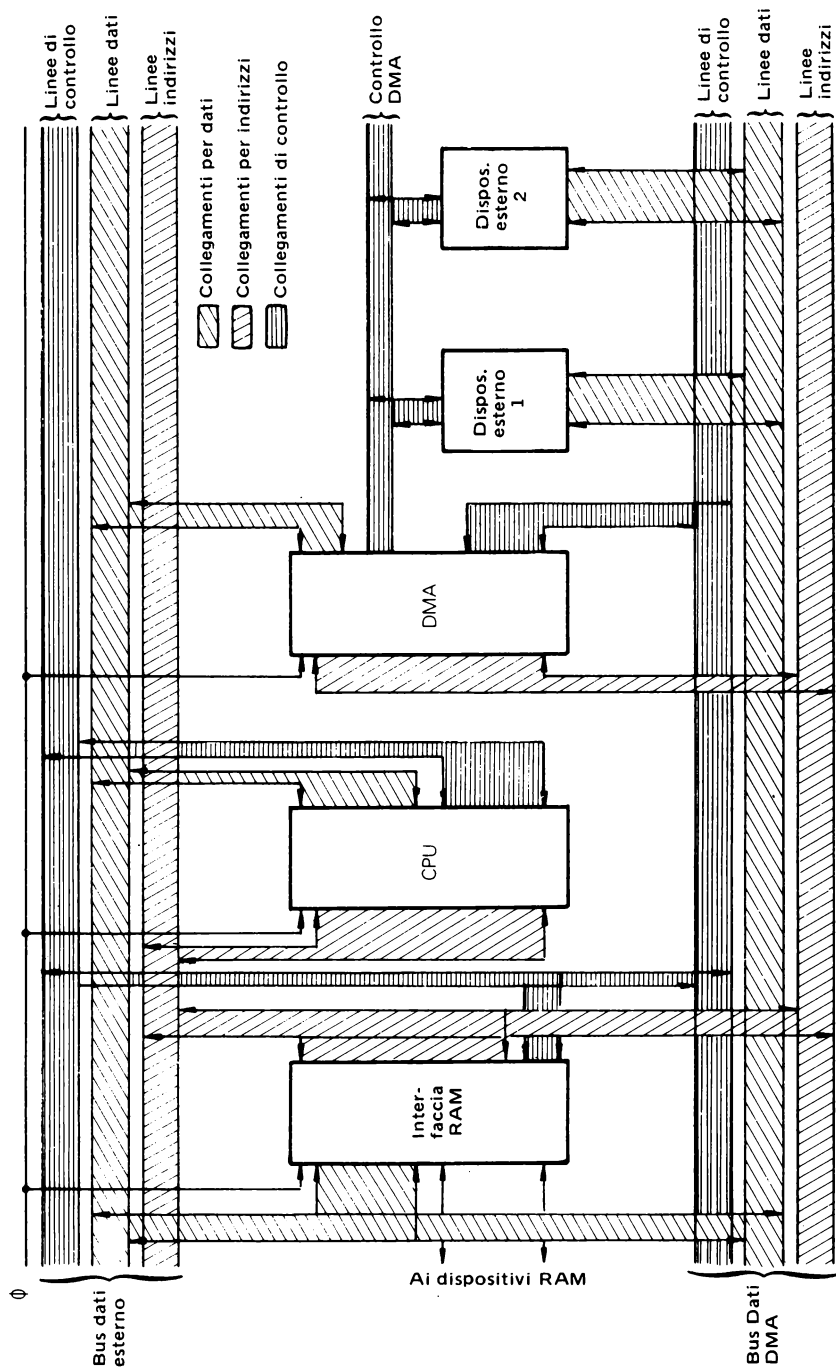



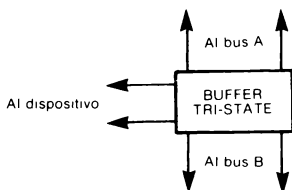
Figura 5-17. Collegamenti per dati, per indirizzi e di controllo utilizzati in DMA simultanei

Notate che anche se il bus del sistema esterno è costantemente occupato, nè la memoria nè i dispositivi esterni saranno occupati quando Φ si trova a 1 durante il ciclo di istruzioni. Questi periodi, (segnati in scuro nella figura) rappresentano il tempo richiesto dall'unità di controllo della CPU per generare i segnali di controllo appropriati.

Creando un secondo bus del sistema esterno, la logica DMA può accedere ai moduli di memoria, mentre la CPU non lo fa. Lo vedete illustrato nella Figura 5-17.

I dispositivi esterni saranno collegati alle linee dati del bus del sistema DMA, come mostra la Figura 5-17, per tutti i trasferimenti dati DMA. Se i dispositivi esterni accedono anch'essi al microcomputer l'I/O programmato, ci devono essere dei collegamenti aggiuntivi alle linee dati del bus del sistema esterno; questi collegamenti aggiuntivi non sono mostrati nella Figura 5-17.

BUFFER TRI-STATE

Ai moduli di memoria, la medesima memoria e i medesimi pin d'indirizzo devono comunicare con due bus e ciò richiederà una forma di giunzione a T. Tale tipo di giunzione viene chiamato buffer tri-state. Un buffer tri-state, in effetti, non è altro che una giunzione a T di segnale multiplo:

DMA SIMULTANEO CONTRO DMA A SOTTRAZIONE DI CICLI

Che dire dell'economia del DMA simultaneo? Dobbiamo metterci un altro bus di dati esterno e un po' di buffer tri-state, ma guadagnamo un po' di tempo: un ciclo di clock per ogni byte dati trasferito.

In realtà, il costo extra per il buffer tri-state è molto basso; infatti, non è inconcepibile che i moduli di memoria vengano forniti con già dei buffer tri-state all'interno. Stiamo parlando quindi di una spesa aggiuntiva molto piccola per un altrettanto piccolo miglioramento e nell'esecuzione.

Perciò è più probabile che il progettista di un microcomputer selezioni un metodo DMA o l'altro a seconda che si adatti meglio all'architettura del microcomputer.

IL BUS DEL SISTEMA ESTERNO

I segnali del bus del sistema esterno, come illustrato nel corso di tutto il capitolo, non hanno una configurazione standard alla quale tutti i microcomputer devono conformarsi.

Il bus del sistema esterno rappresenta una delle caratteristiche più variabili con i microcomputer. A dire il vero, l'unica caratteristica costante che troverete di bus in bus è la presenza di otto linee dati (per microcomputer a 8 bit). Molti microcomputer hanno anche 16 linee d'indirizzo.

La variazione maggiore si può vedere nei segnali di controllo generati dalla CPU. Fondamentalmente vi sono due concezioni estreme. Un estremo consiste nell'avere un set complesso di segnali di controllo al quale rispondono passivamente altri dispositivi. L'altro estremo consiste nell'avere un set di segnali di controllo elementari che devo-

no essere interpretati dagli altri dispositivi che quindi devono avere una discreta quantità di logica al loro interno.

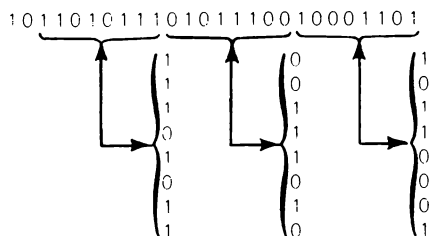
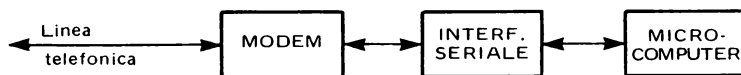
Consideriamo dapprima la CPU chê domina tutto il microcomputer. I dispositivi che interfacciano questa CPU non riceveranno in input nessun segnale di clock. Riceveranno invece numerosi segnali di controllo che identificano dettagliatamente eventi sul bus di dati. I microcomputer della National Semiconductor e della Signetics sono i migliori esempi di questa filosofia.

Il Rockwell PPS-8 rappresenta l'altro estremo. Questo microcomputer mette in uscita solo due segnali di controllo, uno per identificare l'ingresso o l'uscita dei dati. Tutti i dispositivi che fanno da supporto alla CPU PPS-8 ricevono questi due segnali di controllo, più il segnale di clock del sistema. I dispositivi contengono logica interna per decodificare questa combinazione di tre segnali secondo le regole del microcomputer PPS-8.

La filosofia che sta dietro i microcomputer del tipo National Semiconductor e Signetics consiste nel fatto che non occorrono dispositivi speciali per dare un supporto alla CPU. Avendo un set di segnali di controllo veramente completo, si può usare la logica standard.

La filosofica dei microcomputer che hanno set di segnali di controllo molto elementari consiste nel fatto che dato che virtualmente non costa niente aggiungere della logica in più ad un chip LSI, fate molto meglio avere dispositivi che fanno da supporto alla CPU in un modo ben preciso e definito.

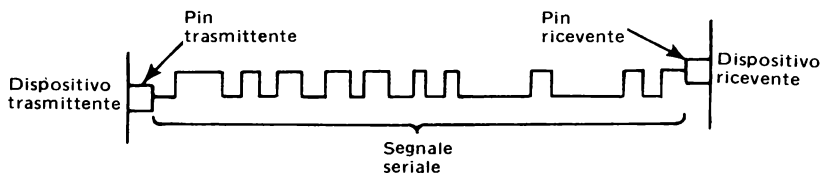
INPUT/OUTPUT SERIALE

I dati vengono trasmessi sulle linee telefoniche in modo seriale. Vi sono anche dei dispositivi di I/O lenti, come le comuni telescriventi e le cassette a nastri magnetici, che trasmettono e ricevono i dati in modo seriale. Se una CPU deve trasmettere o ricevere dati in modo seriale, deve avere una logica di interfaccia capace di convertire i dati seriali in dati paralleli, o i dati paralleli in dati seriali:

Questi sono i passi tramite i quali i dati vengono trasferiti fra una linea telefonica ed un microcomputer:

MODEM **Un modem è un dispositivo che può tradurre i segnali di linea telefonica in livelli logici digitali, o i livelli logici digitali in segnali di linea telefonica.** Alcuni costruttori di microcomputer forniscono i modem come dispositivi

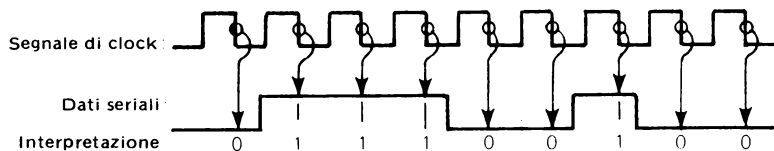
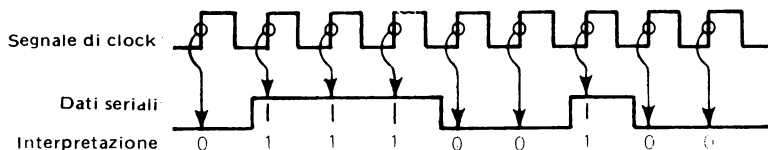
logici, ma poi non li consideriamo come una parte del sistema microcomputer; per questa ragione **i modem non sono descritti in questo libro.**

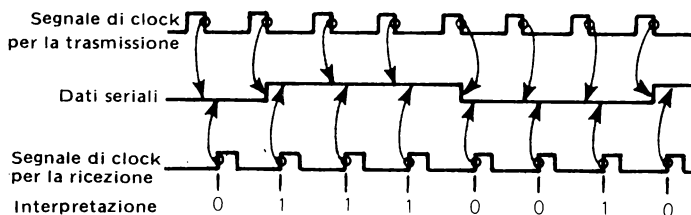
Una cassetta magnetica, o una telescrivente, o qualunque altro dispositivo seriale, possono collegarsi direttamente all'interfaccia seriale:

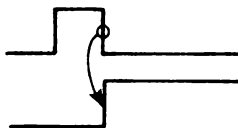
IDENTIFICAZIONE DEI BIT DEI DATI SERIALI

L'unica proprietà di un flusso di dati seriali è che i dati vengono trasmessi e ricevuti come un singolo segnale, per mezzo dei singoli pin dei dispositivi:

Come deve interpretare il dispositivo ricevente un segnale seriale? Come qualunque altro segnale digitale può avere un livello "alto" (+5V) che rappresenta la cifra 1, o un livello basso (0V) che rappresenta la cifra 0.
Consideriamo la sequenza di dati binari: 011100100. Questa è la rappresentazione del suo segnale dati seriali:

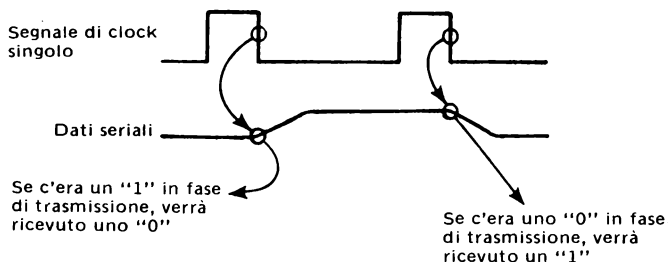
SEGNALI DI CLOCK

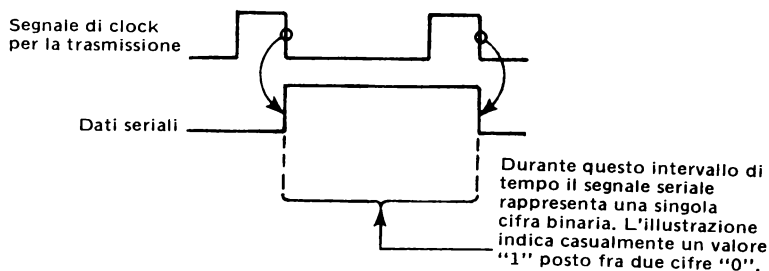
Mentre per voi è facile guardare il segnale dei dati seriali ed interpretarlo dentro le linee verticali tratteggiate, il dispositivo ricevente avrà bisogno di evidenziare maggiormente i limiti di ogni bit del lato. **Useremo un segnale di clock per identificare l'istante in cui il dispositivo ricevente deve interpretare il segnale dati:**

Come illustrato, il fronte di discesa del segnale di clock identifica l'istante in cui il segnale dei dati seriali deve essere campionato. Potremmo altrettanto facilmente prendere il campione sul fronte di salita del segnale:

Il dispositivo trasmittente deve creare un segnale di dati seriali prima che questo possa essere interpretato dal dispositivo ricevente. Vediamo che cosa implica questa necessità.

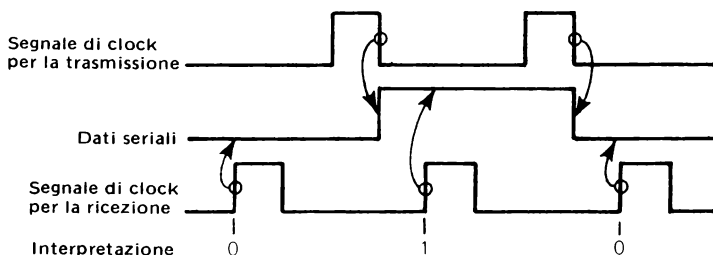
Se il dispositivo ricevente usa un clock per interpretare il segnale dei dati seriali, il dispositivo trasmittente deve usare un clock della stessa frequenza per creare il segnale dei dati seriali:

I segnali di clock del trasmittente e del ricevente non possono essere identici, perchè, in realtà, occorre un tempo finito ad un segnale per cambiare stato.

E' più facile seguire le figure se le transizioni dei segnali sono disegnate come onde.

TEMPO DI STABILIZZAZIONE DEL SEGNALE

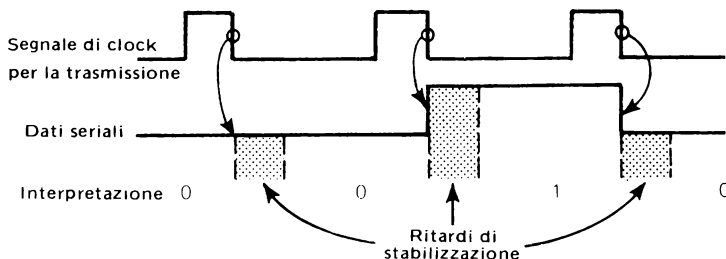
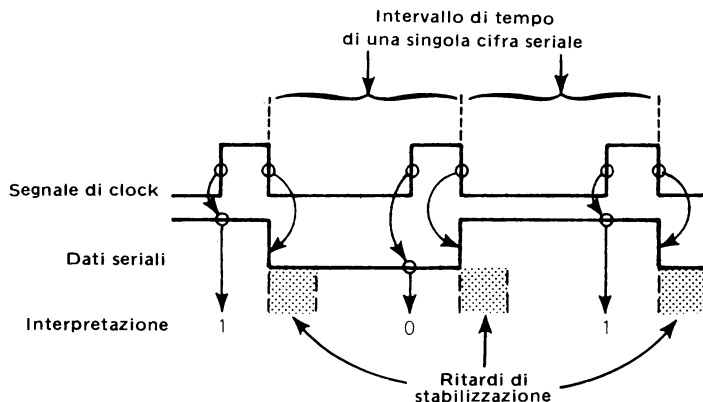
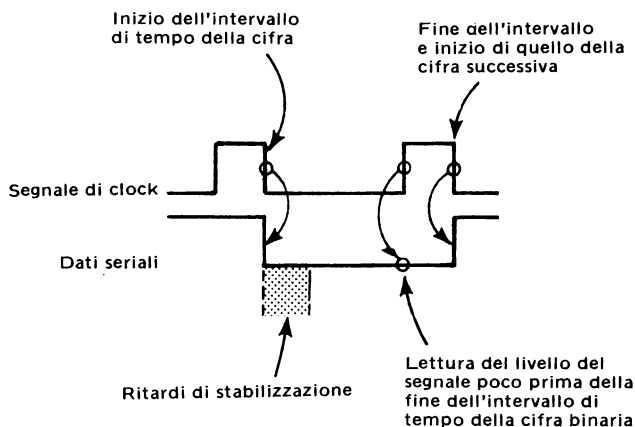
Ma in realtà, ogni segnale che cambia stato richiede un tempo di stabilizzazione finito:

Guardate che cosa succede se usate lo stesso cambiamento di stato di un segnale per trasmettere e ricevere dei dati seriali:

SEGNALI DI CLOCK PER LA TRASMISSIONE DI DATI SERIALI

Concludiamo che i segnali di clock del trasmittente e del ricevente anche se hanno lo scopo in comune, non possono essere il medesimo segnale soggetto ad un'identica interpretazione. Il segnale di clock del trasmittente identificherà la durata di una cifra binaria:

SEGNALI DI CLOCK PER LA RICERCA DI DATI SERIALI

In un certo punto all'interno dell'intervallo di tempo di ogni singola cifra, **il segnale di clock per la ricezione identificherà il livello del segnale dei dati seriali:**

La figura precedente mostra il segnale di clock del trasmettitore attivo sul suo fronte di discesa, mentre il segnale di clock del ricevente è attivo sul fronte di salita: non vi è nessun significato particolare in quest'uso dei fronti del segnale.

RITARDO DELLA STABILIZZAZIONE DEL SEGNALE

Il dispositivo ricevente deve aspettare che il segnale del dato seriale si sia stabilizzato, supposto che abbia cambiato stato, prima di tentare di leggere. Il ritardo nella stabilizzazione del segnale è una caratteristica del dispositivo trasmittente; la durata di questo ritardo è data dal foglio del costruttore che riporta i dati del dispositivo. Ecco un'illustrazione del ritardo della stabilizzazione.

Possiamo usare un solo segnale di clock per trasmettere e ricevere, a patto che trasmettiamo sul fronte di discesa di ogni impulso di clock, o leggiamo sul fronte di salita dell'impulso di clock successivo:

Guardate attentamente come viene ricevuto il livello del segnale trasmesso:

VELOCITA' IN BAUD

L'intervallo di tempo nel quale il segnale del dato seriale rappresenta una sola cifra binaria è direttamente collegato alla velocità alla quale i dati vengono trasmessi. Supponiamo che vengano trasmesse 110 cifre al secondo; questa è una velocità di trasmissione comune. Ogni cifra seriale durerà:

$$\frac{1000000}{110} = 9091 \text{ microsecondi}$$

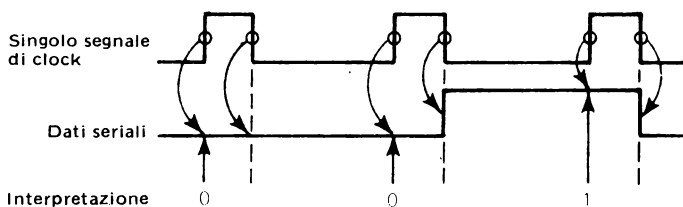
Comunque, la durata di una cifra in un flusso di dati seriali non è il modo in cui vengono misurati i trasferimenti dei dati seriali; **li misuriamo invece in "bit al secondo", e chiamiamo questo numero BAUD.** Per esempio, se vengono trasmesse 110 cifre al secondo, ciò equivale ad un baud rate di 110.

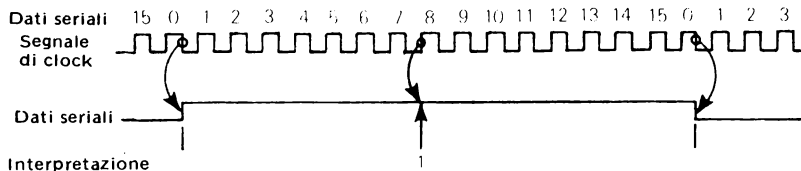
SEGNALI DI CLOCK

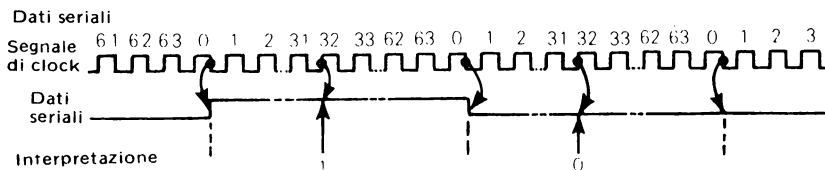
Il nostro microcomputer ha già un segnale di clock, usato per il Timing delle istruzioni all'interno della CPU. **Non confondete il clock del microcomputer con il clock dei dati seriali;** la sola cosa che questi due segnali hanno in comune è che sono entrambi di segnali di clock. **Il segnale di clock dei dati seriali può essere o non essere derivato dal clock del microcomputer.**

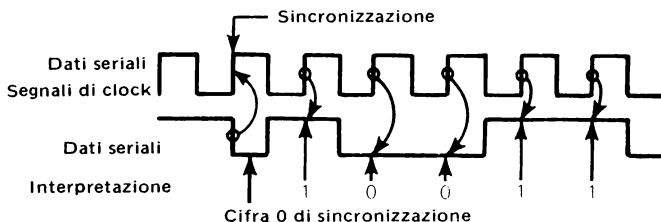
Dal punto di vista dell'utente di microcomputer, la velocità è la differenza più rilevante fra il clock del microcomputer e il clock dei dati seriali. Il clock di un microcomputer tipico può avere un periodo di 500 nanosecondi (2 MHz) mentre le velocità di trasferimento dati seriali vanno tipicamente da 110 a 9600 Baud, 110 Hz a 9,6 KHz. In altre parole, la velocità maggiore di trasferimento dati seriali è circa 200 volte più lenta del clock della CPU di un microcomputer tipico.

SEGNALI DI CLOCK SERIALI x 1

Il clock di I/O seriale non deve necessariamente pulsare esattamente alla baud rate, sebbene lo faccia di frequente:

SEGNALI DI CLOCK SERIALI x 16

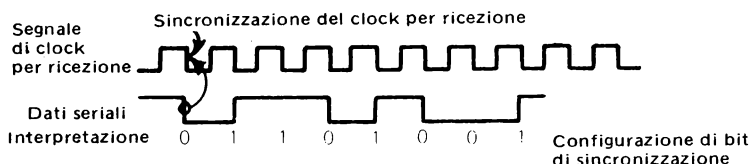
E' molto comune che la velocità del clock sia 16 volte la baud rate:

SEGNALI DI CLOCK SERIALI x 64

Si sceglie spesso anche un clock 64 volte la baud rate:

La ragione per cui si hanno segnali di clock x 16 e x 64, è di poter centrare il più possibile l'intervallo di tempo di ogni singola cifra quando si campiona il segnale dei dati seriali.

Il fatto che i dati seriali abbiano bisogno di un segnale di clock che li accompagni, non significa necessariamente che tutti gli I/O seriali richiedano due linee di segnale. Il segnale di clock di accompagnamento non deve essere trasmesso su di un filo di accompagnamento. Se preparate un'interfaccia per le comunicazioni di dati seriali, con una baud rate predefinita, la logica del dispositivo ricevente non deve ricevere il segnale di clock di accompagnamento. **La logica del dispositivo ricevente può creare il suo segnale di clock locale,** sincronizzandolo con quanto trasmesso nella linea dei dati seriali:

MARKING

Nella figura precedente, il segnale di dati seriali è permanentemente a 1 quando non trasmette dati; questa tecnica viene spesso chiamata Marking.

Notate che quando si usa un segnale di clock x 16 o x 64, il clock che riceve può essere fuori fase di uno o due impulsi rispetto al clock trasmittente senza causare danni. Il punto di campionatura ricevente sarà semplicemente un po' spostato rispetto al centro. Se una sola cifra binaria di sincronizzazione è insufficiente, **cosa ne pensate di una configurazione di cifre di sincronizzazione?**

Possiamo, ad esempio, definire una speciale sequenza di bit di dati seriali e stabilire delle regole che stabiliscano che ogni flusso di dati seriali deve essere preceduto da questa configurazione di sincronizzazione:

CARATTERE SYNC
PROTOCOLLO NEI DATI SERIALI

La configurazione di sincronizzazione illustrata nella figura precedente esiste, nella forma illustrata, e **viene chiamata carattere SYNC.**

Specificare che un flusso di dati seriali deve usare le cifre o i caratteri di sincronizzazione è la prima delle molte **regole che dobbiamo imporre ai flussi di dati seriali al fine di assicurarsi che il dispositivo ricevente interpreti correttamente i dati trasmessi. Questo insieme di regole è conosciuto come "protocollo di comunicazione".** Ogni gruppo di dati di I/O seriali deve avere un protocollo di comunicazione, dato che i dati seriali devono essere completamente autodefiniti. Al contrario dell'I/O parallelo, la linea dei dati seriali non può essere sempre accoppiata alle linee di controllo che dicono in ogni istante al dispositivo ricevente come interpretare i dati.

LINEE TELEFONICHE

Quando avete a che fare specificatamente con delle linee telefoniche, considerate il fatto che i dispositivi trasmittenti e riceventi possono continuamente commutare i ruoli, come accade in qualunque conversazione telefonica. Mentre parlate, voi siete il trasmittente; mentre ascoltate, siete il ricevente.

Analogamente quando si trasmettono dati seriali su linee telefoniche sarà quasi sempre necessaria la comunicazione nei due sensi.

HALL DUPLEX

FULL DUPLEX

Se si usa una sola linea telefonica per trasmettere i dati in entrambe le direzioni, si dice che la comunicazione è hall duplex. Se due linee telefoniche collegano i dispositivi trasmittenti e riceventi, ed ogni linea viene dedicata al trasferimento dei dati in una sola direzione, **si dice che la comunicazione è full duplex**. Il vantaggio della comunicazione telefonica full duplex è che il trasferimento dei dati in etrambe le direzioni può avvenire in parallelo.

RILEVAZIONE DEGLI ERRORI

Sia che i dati seriali siano trasmessi su linee telefoniche, o direttamente fra un dispositivo trasmittente ed uno ricevente, dobbiamo controllare che non ci siano errori nella trasmissione. Se i segnali dati spurii vengono trasmessi sulla linea dei dati seriali, il dispositivo ricevente deve avere dei mezzi per determinare se nei dati si siano inseriti degli errori.

BIT DI PARITA'

Ad un livello abbastanza primitivo, questo compito è assolto dal bit di parità. Dato che il bit di parità è stato settato o risettato, per assicurare che il numero totale di 1 bit a 1 dell'unità dati sia o dispari o pari, sarà rilevato un errore ogni volta che ha un numero dispari di bit sbagliati. Ecco alcuni esempi, con la parità dispari; in tutte le illustrazioni, il bit di parità è su fondo scuro e i bit errati sono contrassegnati con un asterisco:

Trasmesso	Ricevuto	
101101100	100101100 *	Parità pari, individuato errore
101101100	110101100 **	Parità dispari, nessun errore
100101101	100101100 *	Parità pari, individuato errore
100101101	011010011 *******	Parità dispari, nessun errore

CARATTERE DI RIDONDANZA CICLICA

Un'altra tecnica usata per controllare gli errori nella trasmissione è quella di aggiungere un "carattere di ridondanza ciclica" alla fine dei segmenti del flusso di dati. Il carattere di ridondanza ciclica è un numero creato dividendo il flusso di dati trasmesso per un polinomio prefissato. Ecco un divisore di 17 cifre binarie comunemente usato:

11000000000000101

Il risultato di questa divisione nel flusso di dati trasmesso, considerando quest'ultimo come un numero binario continuo, diventa il Carattere di Ridondanza Ciclica. Il dispositivo ricevente moltiplica il flusso di dati ricevuti per il carattere di ridondanza ciclica. Se il risultato non è il divisore standard, allora deve esserci un errore.

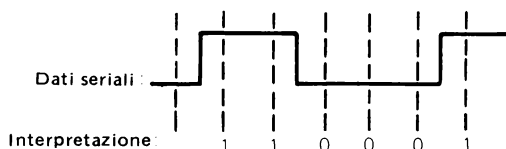
Il carattere di ridondanza ciclica è un metodo abbastanza semplice usato per rintracciare gli errori nella trasmissione. Sono stati elaborati metodi molto complessi, non solo per il ritrovamento degli errori, ma anche per determinare esattamente di che errore si tratta — in modo che possa essere corretto. Sono stati scritti libri interi sull'argomento della rilevazione e correzione degli errori, e quindi non ne parleremo oltre.

PROTOCOLLO DI INPUT/OUTPUT SERIALE

Riuniamo insieme le diverse caratteristiche del trasferimento dei dati seriali che sono state fin qui descritte.

Come affermazione generale, il protocollo di comunicazioni di dati seriali può essere diviso in categorie sincrone e asincrone. Vi sarà più semplice capire il protocollo se considerate le comunicazioni di dati sincroni e asincroni come due entità separate e distinte — non sottovariazioni di un solo concetto.

TRASFERIMENTO SINCRONO DI DATI SERIALI

La caratteristica principale del trasferimento di dati seriali sincrono, è che i dati sono esattamente conformati ad un segnale di clock. Una volta stabilita la baud rate di un trasferimento di dati seriali, il dispositivo trasmittente DEVE trasmettere un bit dati ad ogni impulso di clock; perciò il dispositivo ricevente sa esattamente come interpretare il segnale dei dati seriali:

Per esempio, se è stato specificato un trasferimento di dati seriali sincrono di 300 baud, il dispositivo ricevente può suddividere il segnale dei dati seriali in segmenti di 3333 microsecondi, interpretando ogni segmento come una singola cifra binaria.

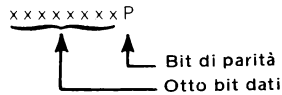
Abbiamo già illustrato segnali di clock con frequenze che sono 1, 16 o 64 volte la baud rate. I segnali di clock x 16 e x 64 potrebbero in teoria essere usati con I/O seriali sincroni ma in pratica non lo sono.

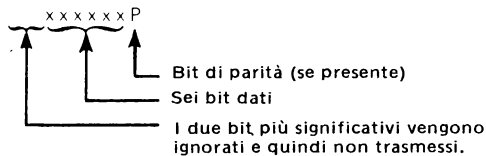
In che modo il dispositivo ricevente può conoscere l'inizio e la fine di ogni unità dati?

Flusso di dati seriali. ··01100101110110001011 1--

Dove sono i limiti del byte?

Chiaramente il nostro protocollo sincrono deve definire la lunghezza delle unità dati individuali — e deve fornire al dispositivo ricevente un modo per sincronizzarsi sui confini dell'unità dati. Il carattere SYNC viene usato a questo scopo. **Ogni flusso di dati sincrono inizia o con uno o con due caratteri SYNC:**

L'unità dati in un flusso di dati seriali sincrono consiste di solito di bit di dati senza parità; ma può anche essere presente un bit di parità. Ecco un esempio di un'unità dati di 9 bit; otto bit di dati ed un bit di parità:

Può essere usata la parità dispari o quella pari.

Non sempre è necessario che vengano trasmessi otto bit di dati; a scelta possono essere 5, 6, 7 o 8 i bit di dati che hanno un significato. Se i bit di dati significativi sono meno di otto, i bit di ordine superiore sono ignorati. Ecco un esempio di un'unità dati in cui solo sei bit di dati sono significativi:

"HUNT MODE" SERIALE SINCRONO

In attesa che incomincino ad arrivare i dati seriali sincroni, il dispositivo ricevente inserirà un "hunt mode", durante il quale esso scandisce continuamente l'input dei dati seriali tentando di far coincidere un flusso di dati seriali entranti con la configurazione SYNC standard. Se il vostro protocollo richiede un carattere SYNC, il dispositivo ricevente incomincerà ad interpretare i dati non appena verrà incontrata una configurazione SYNC. Più spesso, il protocollo richiederà due caratteri SYNC iniziali, nel qual caso il dispositivo ricevente non incomincerà a decodificare i dati finchè non incontra due caratteri SYNC, uno dopo l'altro.

Abbiamo già stabilito che la trasmissione di dati sincrona richiede che il dispositivo trasmittente mandi continuamente dati. Che succede se il dispositivo trasmittente non ha dati da mandare? **In questo caso il dispositivo trasmittente continuerà ad inviare caratteri SYNC finchè un carattere reale non è pronto per essere trasmesso.** Per illustrare questo concetto, **consideriamo un operatore che inserisce dati dalla tastiera;** la tastiera trasmette i dati usando un I/O seriale sincrono, molto lento. L'operatore deve introdurre dalla tastiera il messaggio:

Goodb̸morningb̸Mr.b̸Smith.

b̸ rappresenta il carattere spazio.

Dato che l'operatore inserirà i dati a velocità variabile, la trasmissione dei dati seriali dalla tastiera avrà un carattere SYNC per ogni momento in cui l'operatore lavora troppo lentamente; così il messaggio può essere trasmesso in questo modo:

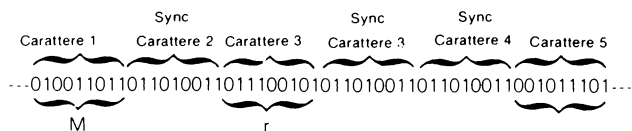
Gos̸os̸s̸ds̸b̸s̸s̸mos̸rns̸ingb̸s̸s̸s̸Ms̸rs̸s̸.s̸b̸s̸s̸s̸Sms̸s̸iths̸

s̸ rappresenta il carattere SYNC.

Quando il dispositivo ricevente decodifica un carattere SYNC nel mezzo di un messaggio, lo ignora, ma resterà in sincronizzazione con il flusso di dati seriali, pronto per interpretare il carattere successivo.

Supponiamo che il messaggio illustrato qui sopra venga trasmesso in codice ASCII e consideriamo questa parte del messaggio:

Mr.

L'equivalente parallelo binario e seriale sincrono può essere illustrato in questo modo:

sono illustrati 9 caratteri e un bit di parità dispari.

Se vengono trasmessi solo caratteri ASCII, si possono usare caratteri a 8 cifre, con 7 cifre di dati ed una cifra di parità.

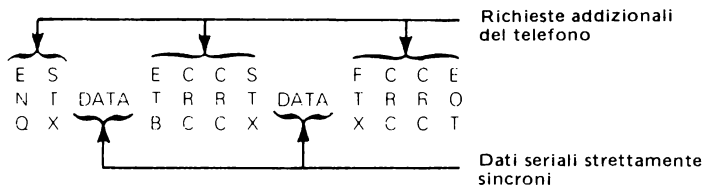
PROTOCOLLO TELEFONICO SINCRONO

In un vero flusso di dati seriali sincrono quando si comunica per mezzo di linee telefoniche, bisogna inserire delle informazioni aggiuntive. Prima di tutto, dobbiamo considerare la necessità di avere una comunicazione nei due sensi.

HANDSHAKING DEI DATI SERIALI
PROTOCOLLO BISINCRONO

Se deve avvenire un collegamento in trasmissione di dati seriali, **il dispositivo trasmittente deve essere in grado di trasmettere comandi e ricevere risposte in colloquio con il dispositivo ricevente; questo è il solo modo di assicurarsi che il dispositivo ricevente sia pronto per i dati trasmessi — o per dire al dispositivo trasmittente che deve smettere di trasmettere e incominciare a ricevere.** Dopo, i caratteri Sync che iniziano e terminano ogni flusso di dati, quindi, di solito vi saranno dei caratteri di controllo ben definiti che devono essere trasmessi in entrambe le direzioni.

Questo dialogo viene chiamato protocollo handshaking. Per esempio, il protocollo Bisync Standard IBM 2770 usa la sequenza di handshaking come in Figura 5A:

All'interno della sequenza delle comunicazioni telefoniche, i DATI saranno trasmessi usando le regole sincrone precedentemente descritte:

TRASFERIMENTO SINCRONO DI DATI SERIALI

Quando i dati seriali vengono trasferiti in modo asincrono, il dispositivo trasmittente trasmette un carattere solo quando ne ha uno pronto da trasmettere. Fra un carattere e l'altro viene messo in output un segnale di "break" continuo, di solito un livello 1 (alto).

Figura 5-A.

Trasmittente

Ricevente

Volendo trasmettere, emettete un carattere "enquiry" per determinare se il ricevente è pronto per ricevere.

Supponendo che sia pronto, il ricevente rimanda un carattere che significa che a ricevuto enquiry ed è pronto a ricevere.

Il trasmittente manda ora un carattere che dice che il testo sta per iniziare. Manda poi un blocco di dati, seguito da un carattere che (1) indica la fine del blocco, e (2) chiede al ricevente di controllare i caratteri di ridonanza ciclica della trasmissione che segue e far sapere al trasmittente se vi sono errori di trasmissione.

Se non sono stati rivelati errori, il ricevente rimanda un carattere diverso; questo dice: "Si, ho ricevuto il tuo blocco, l'ho controllato, è OK, ed ora ho ricevuto da te un numero dispari di blocchi di lettura".

E N Q | S T X DATA ECC TRR BCC | S T X DATA ECC TRR XCC | E O T

Caratteri inviati dal trasmittente

Il trasmittente invia il suo solito carattere "inizio del testo" seguito dal testo. Se questo è l'ultimo blocco di dati che viene trasmesso, viene inviato un carattere di fine testo seguito da un carattere di ridonanza ciclica.

D L 0 E | D L 1 E | D L E O

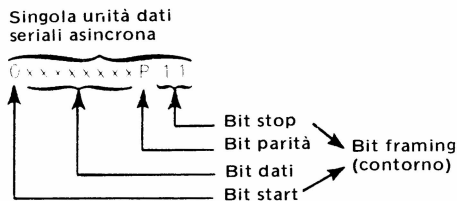
(Caratteri inviati dal ricevente)

Supponendo che il ricevente non abbia trovato errori, rimanda un carattere di riconoscimento che dice: "Il tuo ultimo blocco è stato ricevuto OK ed ora ho ricevuto un numero pari di blocchi di lettura".

Avendo ricevuto la risposta, e non avendo più niente da inviare, il trasmettitore manda semplicemente uno speciale carattere di fine trasmissione.

Riconoscendo lo speciale carattere di fine trasmissione, anche il ricevente toglie il collegamento.

FRAMING
BIT DI START
BIT DI STOP

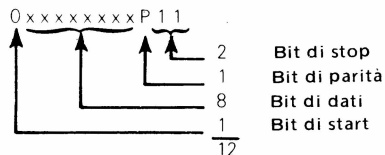
Ogni unità dati di un flusso di dati asincrono deve portare la sua informazione di sincronizzazione. Un'unità dati asincrona è quindi "contornata" (framing) da un solo bit di start, e da uno, uno e mezzo, o due bit di stop:

Il fatto di avere un solo bit di start a 0 è universalmente accettato nel mondo dei microcomputer.

Vi è un'analogia fra i caratteri SYNC del flusso di dati sincrono e i bit di framing di un flusso di dati asincrono.
I caratteri SYNC delimitano un blocco di caratteri di dati sincroni. I bit di start e di stop delimitano ogni carattere dei dati in un flusso di dati asincrono.

Degli otto bit di dati, 5, 6, 7 o 8 possono essere significativi, come nel caso dei dati seriali sincroni. Se i bit significativi sono meno di otto, i bit di ordine superiore, più a sinistra, sono ignorati. Per esempio, se il vostro protocollo stabilisce che ci saranno solo cinque bit di dati in ogni parola asincrona trasmessa, il dispositivo ricevente riceverà solo cinque bit di dati, ed interpreterà ogni parola ricevuta nel modo seguente:

Un'unità dati a 9 bit, in realtà, viene cosi trasmessa.

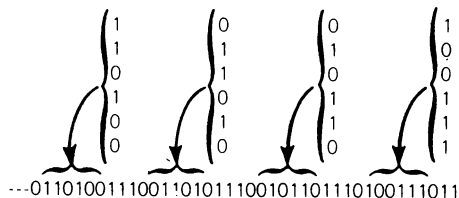
BIT DI PARITA'
BIT DI STOP

Il bit di parità è sempre presente. Può essere specificata la parità dispari o quella pari.
Per i bit di stop vengono sempre usati gli 1. Più spesso ci sono due bit di stop; uno di questi viene qualche volta specificato.
Se avete due bit di stop, ogni parola di dati seriali di 8 bit conterrà dodici bit:

Se avete un bit di stop, ogni parola di dati seriali di 8 bit consisterà di undici bit.

FORMATO DEI DATI SERIALI DELLA TELESCRIVENTE

Le telescriventi usano un bit di start, sette bit di dati, un bit di parità e due bit di stop — per un totale di 11 bit per carattere. Le telescriventi operano ad una velocità standard di 10 caratteri al secondo, che si traduce in 110 Baud.

Alcuni protocolli di trasmissione hanno un bit e mezzo di stop. La larghezza del bit di stop è una volta e mezza la larghezza del bit normale.

Consideriamo i dati seriali asincroni, di parità pari, che usano due bit di stop, con 6 bit di dati in ogni unità dati. Ecco come una sequenza di dati paralleli sarà convertita in un flusso di dati seriali:

Se le comunicazioni di dati seriali sincrone avvengono su linee telefoniche, sarà necessaria una forma di protocollo di handshaking, come abbiamo illustrato per le comunicazioni telefoniche sincrone. Infatti, non c'è nulla che vieti di usare lo stesso protocollo di handshaking. Questo protocollo è semplicemente un metodo per trasmettere le informazioni fra due dispositivi per mezzo di una sola linea telefonica.

ERRORE DI FRAMING

Notate che durante il trasferimento dei dati asincroni il dispositivo ricevente ha un mezzo in più per controllare gli errori di trasmissione. La prima cifra binaria di ogni unità dati deve essere uno 0 che rappresenta il bit di start; le ultime due cifre binarie dell'unità dati devono essere entrambe 1, che rappresentano i bit di stop. **Se il dispositivo ricevente non rileva i bit di start e di stop appropriati per qualunque unità dati in un flusso di dati seriale asincrono, allora denuncerà un errore di framing.**

DISPOSITIVO DI COMUNICAZIONI DI I/O SERIALE

Vediamo ora che cosa occorre ad un dispositivo di interfaccia di I/O seriale.

MISURE DEL DUAL IN-LINE PACKAGE

Prima di tutto, quanto dovrebbe essere grande il DIP? Abbiamo usato DIP da 40 pin indifferentemente per tutti i nostri dispositivi. **Vi è una ragione che ci porti ad usare una misura di package più grossa o più piccola, o facciamo meglio a standardizzare semplicemente il DIP da 40 pin, anche se metà dei pin restano inusati?**

La risposta è che, restando uguale tutto il resto, sarebbe meglio usare dei DIP con meno pin possibili. I DIP più grossi costano di più e occupano più spazio sulla scheda di un circuito stampato. Usando un DIP da 40 pin, dove si potrebbe usarne uno più piccolo, si ottiene un effetto "palla di neve" per quanto riguarda i costi: meno DIP su una scheda di circuito stampato possono significare più schede di circuito stampato. Più schede di circuito stampato possono voler dire un telaio più grande, una maggiore alimentazione ed un involucro più costoso. D'altra parte, non ha senso dal punto di

vista economico avere un'incredibile varietà di misure dei DIP semplicemente per essere sicuri che nessun DIP sprechi mai un pin.
Per esempio, è meglio usare un DIP standard da 40 pin piuttosto che costruire un insolito prodotto da 38 pin.

Per il nostro dispositivo di comunicazioni di I/O seriale sceglieremo un DIP da 28 pin, dato che questa è una delle misure del package. Possiamo cavarcela con questo numero minore di pin perchè le nostre porte di I/O seriale si limiteranno ad 1 pin per ogni porta.

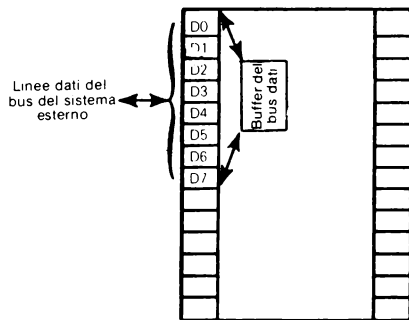
DISTRIBUZIONE DELLA LOGICA

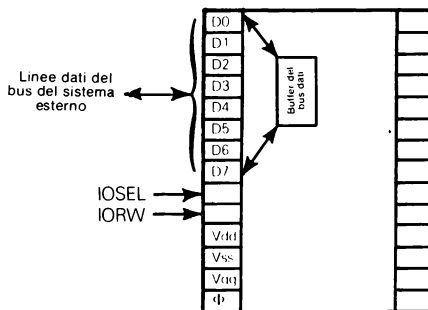
La logica di I/O seriale sincrona ed asincrona staranno su un solo chip. I due set di logica hanno abbastanza cose in comune perchè ciò abbia senso.

Il nostro dispositivo di comunicazioni seriali di I/O può essere visualizzato come se avesse tre interfacce: una per la CPU del microcomputer ed una per ogni I/O seriale, asincrono e sincrono. Ogni interfaccia, come al solito, avrà delle linee di dati e dei segnali di controllo. Per l'interfaccia di I/O seriale, i segnali di controllo possono essere raggruppati in controlli generali e controlli del modem. I controlli generali si applicano ad ogni logica esterna, mentre i controlli del modem soddisfano le necessità specifiche del modem standard dell'industria — il che non vi impedisce, se potete farlo, di usare i controlli dei modem per dell'altra logica esterna.

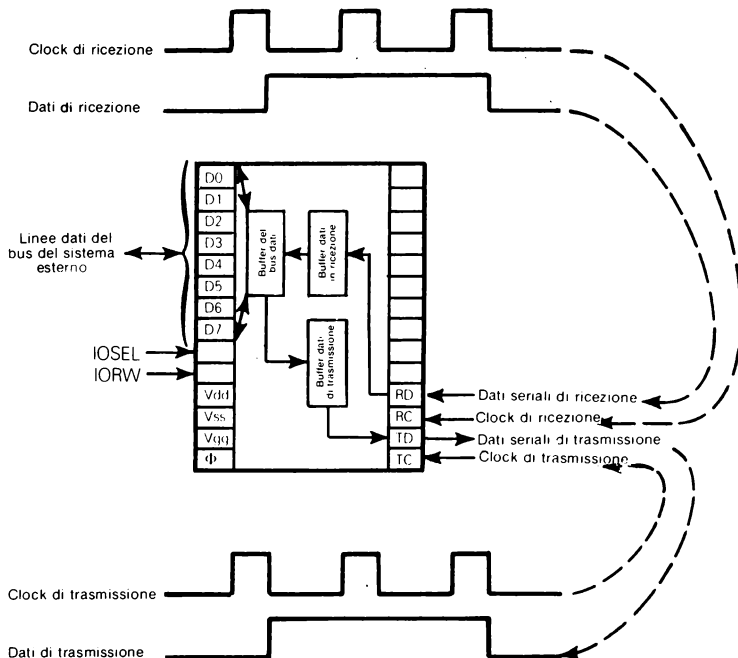
LA CPU — INTERFACCIA DEL DISPOSITIVO DI I/O SERIALE

Dato che l'interfaccia della CPU è comune all'I/O sincrono ed asincrono, incominceremo da qui.

Il dispositivo di I/O seriale comunicherà in parallelo con la CPU tramite le linee dati del bus del sistema esterno. Dobbiamo quindi fornire 8 pin di dati, supportato da un buffer del bus dati:

Gli altri segnali richiesti dall'interfaccia della CPU non differiscono da quelli che abbiamo incluso nel dispositivo di interfaccia di I/O in parallelo. Più precisamente, essi **sono IOSEL e IORW**. Iosel identifica un'operazione di I/O in corso e IORW seleziona o una lettura dalla o una scrittura nella CPU.

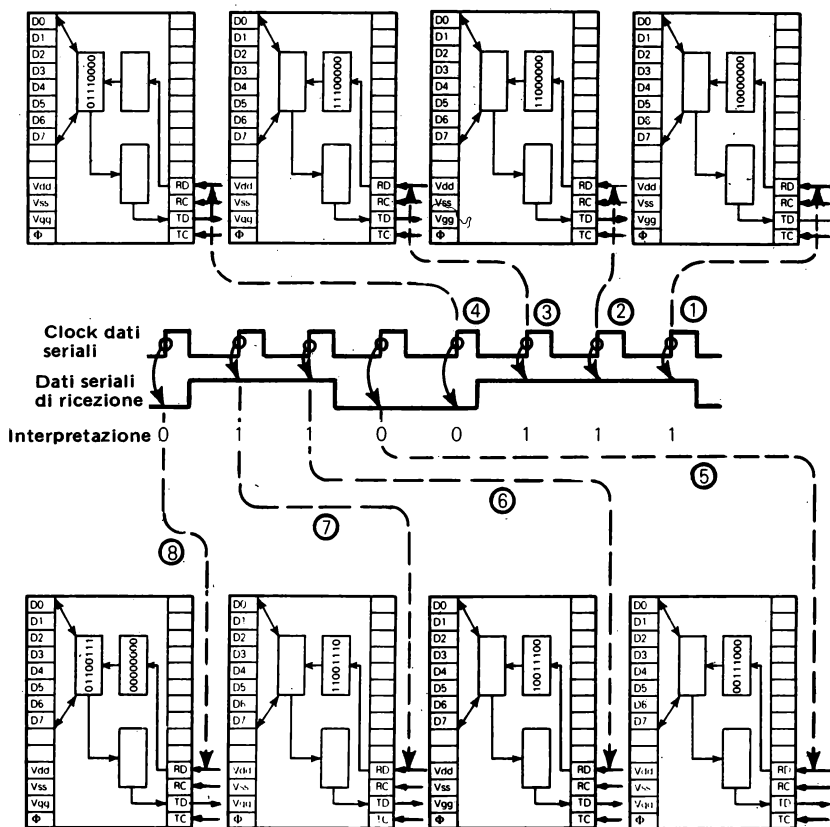
Aggiungeteci il clock, la massa e l'alimentazione, e il nostro dispositivo di interfaccia di I/O seriale appare in questo modo:

L'INTERFACCIA DI I/O SERIALE

Useremo pin separati per trasmettere e ricevere i dati seriali. Alcuni dispositivi usano un solo pin per dati bidirezionali.

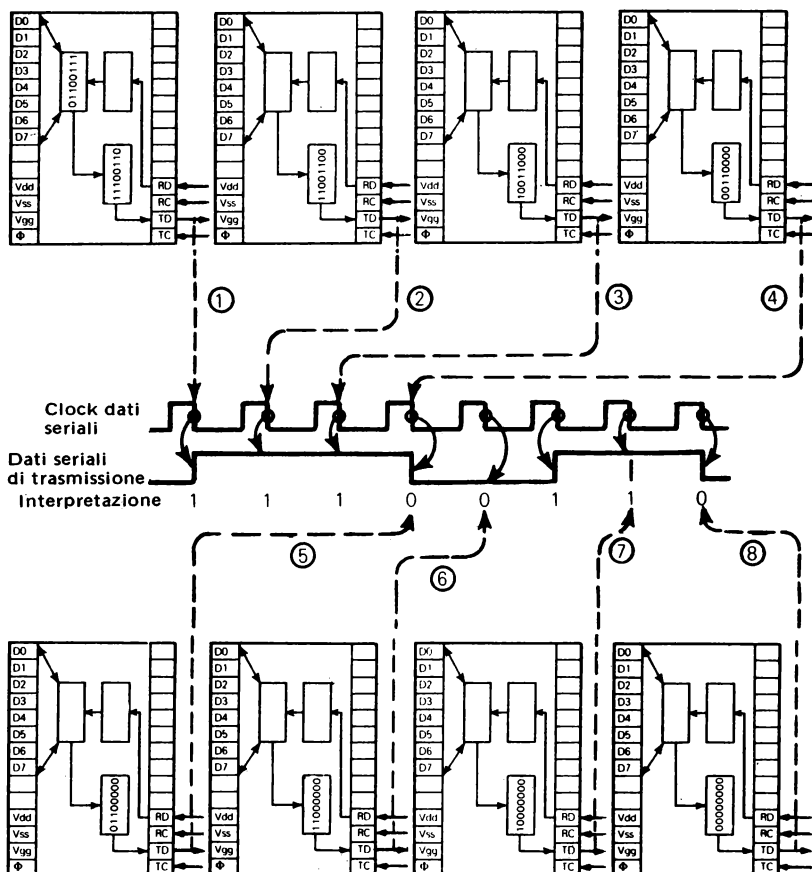
Dato che abbiamo pin di dati di trasmissione e di ricezione separati, **avremo bisogno anche di prendere in input segnali di clock di trasmissione e di ricezione separati.** Entrambi i segnali di clock sono presi dalla logica esterna per controllare la velocità alla quale i dati vengono trasmessi o ricevuti:

SEGNALI DI CLOCK

Se la logica esterna usa lo stesso segnale di clock per la trasmissione e la ricezione, potete derivare entrambi i segnali dal clock del sistema Φ o da qualunque altra logica di clock. **In ogni caso questi due segnali di clock controlleranno la baud rate dei dati seriali.** La baud rate non sarà determinante dalla logica interna al dispositivo di interfaccia di I/O seriale e la logica del dispositivo non genererà o metterà in uscita segnali di clock.

INPUT DEI DATI SERIALI

Consideriamo un flusso di dati sincrono in cui ogni fronte di salita del segnale di clock di ricezione fornirà impulsi di strobe al livello del segnale dei dati di ricezione, come una cifra binaria, nel buffer dei dati di ricezione. **In ogni momento il buffer dei dati di ricezione contiene otto cifre binarie, il suo contenuto sarà trasferito nel buffer del bus dati.** Ecco un'illustrazione dell'input dei dati seriali:

Il buffer dei dati di ricezione è ora pieno, per cui il bit dei dati di ricezione successivo ricomincerà di nuovo il processo di caricamento.

OUTPUT DEI DATI SERIALI

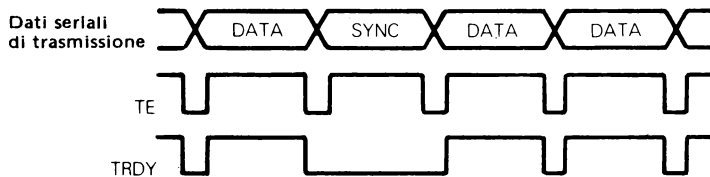
Ogni fronte di salita dell'impulso del clock di trasmissione fornirà impulsi strobe ad un bit del buffer dei dati di trasmissione. **Gli otto bit del buffer dei dati di trasmissione saranno messi in output in ordine ascendente partendo dal bit 0.** Non appena il bit 7 viene messo in output, il buffer dei dati di trasmissione sarà considerato vuoto, così il contenuto del buffer del bus dati sarà caricato nel buffer dei dati di trasmissione, per continuare il processo di trasmissione seriale. Ecco un'illustrazione dell'output dei dati seriali:

Se fossero trasmessi dati seriali asincroni, la relazione fra i segnali di clock dei dati e dei dati seriali cambierebbe, ma questo è tutto. Ricordate che in un flusso di dati asincrono usate un clock x 16 o x 64 e campionate i dati sull'ottavo o sul trentaduesimo impulso – al centro dei bit di dati.

SEGNALI DI CONTROLLO DELL'I/O SERIALE

Il buffer del bus dati non può essere usato simultaneamente per ricevere i byte di dati assemblati e per trasmettere i byte di dati per il disassemblaggio. **La logica di controllo e i segnali di controllo che stiamo per descrivere ora determinano quale delle operazioni possibili sta svolgendosi in ogni momento.** Il dispositivo di interfaccia di I/O seriale ignorerà semplicemente il segnale di clock se la logica di controllo interna non è stata programmata per riconoscerlo. Inoltre, il contenuto del buffer dei dati di ricezione andranno semplicemente perduti se il buffer del bus dati non è pronto per ricevere un byte assemblato.

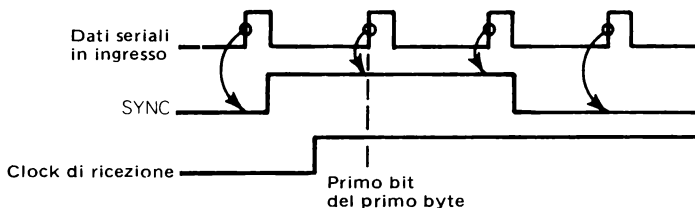
Prendiamo in considerazione i segnali di controllo che devono essere presenti per fare da supporto ai dati seriali che vengono trasmessi e ricevuti.

SEGNALI DI CONTROLLO DELLE TRASMISSIONI SERIALI

Prima di tutto, consideriamo la logica di trasmissione, occorreranno due segnali di controllo, uno per indicare che il buffer dei dati di trasmissione è vuoto, l'altro per indicare che il buffer dei dati è pronto per ricevere un altro byte di dati. Chiameremo questi due segnali TE e TRDY. I due segnali non sono identici. Per esempio, quando i dati seriali vengono messi in output in modo sincrono, TE sarà a 1 mentre viene messo in output un carattere SYNC; TRDY sarà a 0 per indicare che il buffer dei dati di trasmissione è veramente pronto per ricevere un altro byte di dati, anche se i dati vengono in questo momento messi in output. Ecco il modo in cui verranno usati i segnali TE e TRDY:

SEGNALI DI CONTROLLO DELLA RICEZIONE SERIALE

La logica di ricezione usa un solo segnale di ricezione pronta che chiameremo RRDY. Questo segnale dice alla CPU che un byte di dati è stato caricato nel buffer del bus dati e può ora essere letto. **Il segnale RRDY verrà spesso usato per generare una richiesta d'interruzione.** L'interrupt può essere riconosciuto da una semplicissima sequenza di istruzioni che sposta il byte di dati ricevuto in un'appropriata posizione di memoria di lettura/scrittura del microcomputer.

CONTROLLO DELLA SINCRONIZZAZIONE NELLA RICEZIONE SERIALE

Quando vengono ricevuti dati sincroni, ricordate che la logica del dispositivo di interfaccia di I/O seriale deve rilevare uno o due caratteri SYNC prima di riconoscere i dati validi. La logica esterna deve sapere quando il dispositivo di I/O seriale ha rilevato questi caratteri SYNC. **Aggiungeremo quindi un segnale di controllo SYNC, che sarà messo in output a 1 non appena vengono rilevati i caratteri SYNC.** Alcuni dispositivi di I/O seriale permettono che la linea di controllo SYNC sia bidirezionale. In questo caso, invece di precedere i dati sincroni con i caratteri SYNC, **la logica esterna può inserire il segnale di controllo SYNC a 1;** allora il dispositivo di I/O seriale usa questo impulso di controllo allo scopo di iniziare la ricezione dei dati sincroni:

SEGNALI DI CONTROLLO DEL MODEM

Restano da descrivere solo i segnali di controllo del modem. Vi sono quattro segnali di controllo dei modem standard industriali:

1) $\overline{DSR}$ (Data Set Ready) — Il modem mantiene questo segnale a 0 in ogni momento in cui è pronto per ricevere i dati. Il segnale è messo a 1 negli altri momenti. Qualunque altra logica esterna può usare questo segnale come abilitatore/disabilitatore principale. Per esempio, questo segnale potrebbe essere generato da un interruttore aperto/chiuso in una unità come un terminale video esterno. Ciò permette al microcomputer di controllare la logica esterna prima di tentare di comunicare con essa.

2) $\overline{DTR}$ (Data Terminal Ready) — Questo segnale di controllo è l'equivalente di $\overline{DSR}$ del dispositivo di I/O seriale; viene messo in output dal dispositivo di interfaccia di I/O seriale per dire alla logica esterna che è pronto per comunicare. Sotto il controllo del programma potete posizionare questo segnale a 1 per inibire tutte le operazioni di I/O, o a 0 per dare inizio alle operazioni di I/O seriali.

3) $\overline{RTS}$ (Request To Send) — Quando il dispositivo di I/O seriale è pronto per comunicare con un modem o con un'altra logica esterna, sia $\overline{DSR}$ che $\overline{DTR}$ saranno a 0. Ora il dispositivo di I/O seriale usa il segnale RTS per indicare che è pronto per

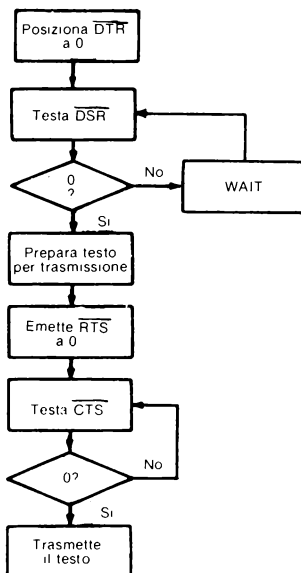
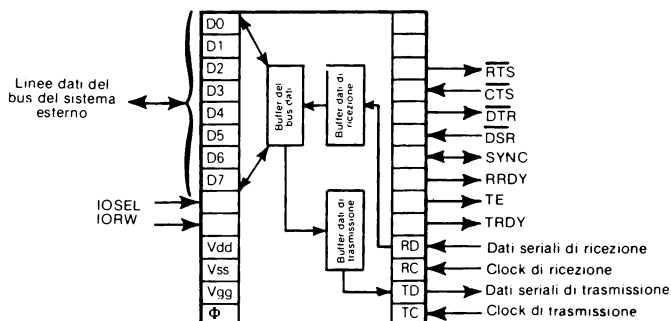



Figura 5-B.

trasmettere i dati. Ricordate che il dispositivo ricevente può essere temporaneamente occupato, anche se è stato attivato.

4) $\overline{\text{CTS}}$ (Clear To Send) — In un collegamento di dati full duplex, $\overline{\text{RTS}}$ del trasmittente diventa $\overline{\text{CTS}}$ al ricevente; se viene inviato $\overline{\text{RTS}}$, chiaramente deve esserci un modem alla fine della linea in grado di riceverlo. In un collegamento di dati half duplex, il modem che riceve $\overline{\text{RTS}}$ rimanda indietro $\overline{\text{CTS}}$ due millisecondi più tardi.

L'interazione di $\overline{\text{DSR}}$, $\overline{\text{DTR}}$, $\overline{\text{RTS}}$ e $\overline{\text{CTS}}$ può essere illustrata dal diagramma di flusso del programma (vedi Figura 5B).

Ecco come appare ora il nostro dispositivo di interfaccia seriale:

IL CONTROLLO DEI DISPOSITIVI DI INTERFACCIA DELL'I/O SERIALE

Date le molte scelte disponibili nell'uso dei dispositivi di interfaccia dell'I/O seriale, avremo bisogno di un registro di controllo per selezionare le possibili opzioni — e in alcuni casi determinare le condizioni dei segnali di controllo che vengono messi in output.

I/O SERIALE

Per prima cosa dobbiamo selezionare gli I/O sincroni o asincroni; poi la Tabella 5-1 identifica le decisioni fondamentali che dobbiamo prendere sotto il controllo del programma. **Ci riferiamo alle variabili della Tabella 5-1 come ai parametri di modo,** dato che è improbabile che vengano cambiati nel corso di una qualunque operazione di I/O seriale.

Tabella 5-1. Parametri di modo di I/O seriale

FUNZIONE	ASINCRONO	SINCRONO
Frequenza del clock	Baud rate x 1,x16 x 64	Di solido baud rate x1
Bit dati per byte	5, 6, 7 o 8	5, 6, 7 o 8
Parità	Dispari, pari o nessuna	Dispari, pari o nessuna
Bit di stop	1, 1 $\frac{1}{2}$ o 2	Non si usa
Caratteri SYNC	Non si usa	1, 2 o SYNC esterno

I/O SERIALE ISOSINCRONO

L'I/O asincrono con l'uso di un clock x 1 viene talvolta chiamato I/O isosincrono; equivale a trasmettere i dati usando il formato dei caratteri asincroni (compresi i bit di framing) in un flusso di dati sincrono diverso.

COMANDI DELL'I/O SERIALE

All'interno di un qualunque set selezionato di parametri di modo, il dispositivo di interfaccia di I/O seriale deve ancora ricevere dei comandi. I comandi devono identificare la direzione del flusso dei dati seriali (trasmissione o ricezione), o terminare le operazioni in corso, permettendo al modo di essere modificato. I comandi devono anche posizionare la condizione dei segnali di controllo $\overline{DTR}$ e $\overline{RTS}$, e rispondere a una qualunque condizione di errore.

CONDIZIONI DI ERRORE DELL'I/O SERIALE

Che cosa sono le condizioni di errore di cui i comandi si devono occupare, e come deve rilevarle il microcomputer? **Forniremo il dispositivo di interfaccia di I/O seriale di un registro di stato di 8 bit.** Avendo 8 bit, possiamo leggere una combinazione di otto stati di segnali di input e condizioni di errore. **I segnali di input di cui dobbiamo essere in grado di leggere il livello sono:**

SEGNALI DI CONTROLLO DELL'INPUT DELL'I/O SERIALE

1) $\overline{DSR}$ - Set di dati pronto. (Date set ready)
2) $\overline{CTS}$ - Azzeramento per invio. (Clear to Send). Questo segnale viene talvolta lasciato fuori dal registro di stato, la logica del dispositivo di interfaccia di I/O seriale deve allora automaticamente aspettare che $\overline{CTS}$ sia vero prima di iniziare un trasferimento di dati seriali.
3) SYNC - Sincronizzazione esterna.
4) TE - Buffer di trasmissione vuoto. (Transmit buffer empty)
5) TRDY - Buffer di trasmissione pronto per ricevere i dati dalla CPU. (Transmit buffer ready)
6) RDY - Buffer di ricezione pronto per inviare i dati alla CPU. (Receive Buffer Ready). Questo segnale può essere collegato alla logica dell'interrupt e lasciato fuori dal registro di stato.

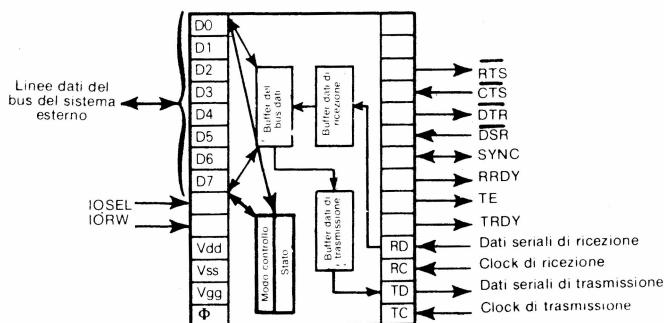
Queste sono le condizioni di errore che possono venire riportate:

1) Errore di parità. La parità sbagliata è stata rilevata in un'unità di dati seriali.
2) Errore di overrun. I buffer di ricezione dei dati ha trasmesso un byte di dati al buffer del bus dati, che non era pronto per ricevere i dati. I dati sono andati perduti.
3) Errore di overrun. I buffer di ricezione dei dati ha trasmesso un byte di dati del bus dati, che non era pronto per ricevere i dati. I dati sono andati perduti.

Normalmente una condizione di errore non fa sì che un dispositivo di interfaccia di I/O seriale abortisca le operazioni. L'errore viene riportato nel registro di stato e le operazioni continuano tranquillamente. **Usando i comandi, possiamo reagire ad una condizione di errore in uno dei seguenti modi:**

1) In modo sincrono, rimandare indietro un carattere di NAK (nessun riconoscimento) alla sorgente della trasmissione.
2) In modo asincrono, abortire le operazioni e posizionare TD al suo livello di segnale "break" (di solito alto).
3) Eseguire un qualunque altro programma di ripristino.

4) Risettare un qualunque bit di errore nel registro di stato del dispositivo di interfaccia di I/O seriale.

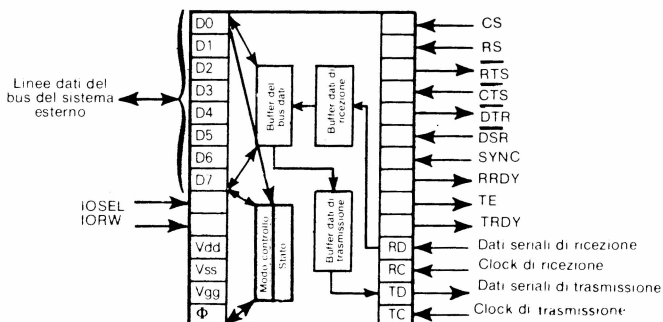
Al nostro dispositivo di interfaccia di I/O seriale dobbiamo ora aggiungere un registro di modo/controllo e un registro di stato:

INDIRIZZAMENTO DEL DISPOSITIVO DI INTERFACCIA DI I/O SERIALE

L'unico aspetto del dispositivo di interfaccia di I/O seriale che non abbiamo esaminato è come viene selezionato il dispositivo, e come vengono indirizzati i suoi buffer e i suoi registri.

Finchè abbiamo a che fare con la logica della CPU, il dispositivo consiste del buffer del bus dati, del registro di controllo e del registro di stato.

I buffer dei dati di ricezione e di trasmissione si trovano passivi nel canale dei dati ricevuti e trasmessi, rispettivamente; comunicano con il buffer del bus dati, perciò non hanno bisogno di un accesso diretto addizionale.

In realtà i registri di controllo e di stato possono essere considerati un'unica unità indirizzabile, dato che potete solo scrivere in un registro di controllo e potete solo leggere da un registro di stato.

Perciò, ci occorrono solo due pin per accedere al dispositivo di interfaccia di I/O seriale — il che va benissimo, perchè abbiamo lasciato liberi solo due pin. Un pin (CS) costituirà la selezione del chip, mentre l'altro pin (RS) seleziona o il buffer del bus dati o il registro di controllo/stato:

Dato che abbiamo solo due pin per indirizzare il dispositivo di interfaccia di I/O seriale, occorrerà della logica esterna per decodificare in modo appropriato le linee d'indirizzo, le linee del bus dati esterno o le linee di controllo, allo scopo di creare i segnali di selezione CS e RS. Nell'industria del microcomputer, l'uso di una logica di selezione di questo tipo costituisce quasi una regola. In realtà, i dispositivi di interfaccia di I/O non avranno 8 linee d'indirizzo, come abbiamo indicato in precedenza in questo capitolo a proposito del dispositivo di interfaccia di I/O in parallelo.

Nella Figura 5-18 possiamo ora illustrare un metodo per integrare un dispositivo di interfaccia di I/O seriale nel nostro ipotetico microcomputer.

LOGICA IN TEMPO REALE (REAL-TIME LOGIC)

Il concetto di logica in tempo reale è abbastanza semplice da capire. L'esempio più ovvio è il mantenimento dell'ora della giornata da parte di un microcomputer che comanda un lettore di badge, e deve quindi registrare l'ora esatta in cui ogni impiegato arriva o lascia il suo posto di lavoro.

Dall'altro capo della scala temporale, un microcomputer può comandare uno strumento che misura la velocità di rotazione di un rullo, di un ventilatore di un propulsore, e calcola la velocità di rotazione in migliaia di giri al minuto.

Un microcomputer non avrà difficoltà a scandire l'ora della giornata dato che tutto il sistema del microcomputer è guidato da un segnale di clock. Tutto ciò che occorre è aggiungere della logica che conteggi i segnali di clock e generi una richiesta d'interruzione dopo un numero specificato di segnali contati. Per esempio, se un segnale di clock ha un periodo di 500 nanosecondi, si potrebbe temporizzare un intervallo di tempo di un millisecondo generando un'interruzione ogni 2000 periodi di clock. Naturalmente, se il segnale di clock del microcomputer verrà usato per misurare l'ora, è necessario un intervallo di tempo preciso. Quando viene usato in un'applicazione che non è sensibile al tempo, un microcomputer può avere un cristallo assolutamente non costoso per generare il segnale di clock.

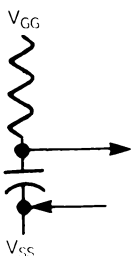
A dire il vero, in alcuni casi, al posto del cristallo si può usare una rete di resistori-condensatori:

Alcuni dei microcomputer descritti nel Volume II hanno un timer programmabile come parte integrante della logica del dispositivo. Quando si usano altri microcomputer, tutto ciò che occorre è una forma di logica di conteggio degli impulsi esterna al microcomputer, che generi una richiesta d'interruzione dopo che è stato conteggiato un numero fissato di periodi di clock.

Figura 5-18. Uso di I/O seriale con interrupt per inviare alla CPU i dati di ricezione

DISTRIBUZIONE DELLA LOGICA FRA I DISPOSITIVI DI UN MICROCOMPUTER

I singoli dispositivi che abbiamo descritto in questo capitolo rappresentano bene la grande quantità di microcomputer che vengono oggi venduti. Comunque, non vi è ragione per cui la logica dovrebbe essere distribuita fra i diversi dispositivi come abbiamo descritto noi; e se guardate ai dispositivi reali descritti nel Volume II, vedrete che vi sono veramente grosse variazioni fra la logica che il costruttore di un microcomputer implementerà su di un chip e quella che implementerà su di un altro.

La chiave del discorso sta nel bilanciamento tra alimentazione e numero di dispositivi; in qualunque momento i costruttori di semiconduttori possono implementare un certo numero di porte nella logica di un singolo chip. Quello che dovrebbe essere questa logica è un affare che riguarda il progettista del chip. E' solo la tradizione che vuole che un'unità centrale e la memoria siano implementate su chip separati. Non vi sono ragioni economiche o scientifiche che definiscano le relazioni fra chip e logica.

Quello che succede in realtà, è che il progettista di un chip incomincia a progettare un'unità centrale. Immediatamente, il progettista si trova davanti ad un'importante modifica; presumendo che la tecnologia sia avanzata al punto tale che egli possa ora mettere il 30% di logica in più sul suo chip rispetto all'ultima volta, che cosa deve essere questo 30% in più? La nuova CPU dovrebbe avere un set di istruzioni più potente con moltissimi modi di indirizzamento tipo quello dei minicomputer, oppure il set di istruzioni dovrebbe restare lo stesso — destinando la logica in più alla memoria di lettura/scrittura? O come fare per usare la logica in più per mettere degli I/O in parallelo sul chip della CPU?

In realtà la quantità di logica che può essere stipata su di un solo chip va aumentando molto rapidamente. E' per questa ragione che probabilmente vedremo un'evoluzione altrettanto rapida nei microcomputer, con orientamenti diversi. **Da una parte dello spettro abbiamo già raggiunto il microcomputer di un singolo chip, dove una versione ridotta di tutti i diversi dispositivi di logica descritti in questo capitolo sono stati riuniti su di un solo chip. Dall'altra parte, stiamo vedendo i microcomputer che riproducono esattamente i minicomputer.**

Concludiamo quindi questo capitolo con un avvertimento: la dispersione della logica su dispositivi diversi, come abbiamo descritto in questo capitolo, non è niente di più di una linea indicativa molto approssimata e col passar del tempo, vedrete che i microcomputer riuniranno la logica su pochissimi dispositivi, o addirittura su uno solo.

Capitolo 6
PROGRAMMAZIONE DEI MICROCOMPUTER

Le istruzioni vengono usate per specificare qualsiasi sequenza logica che può aver luogo all'interno di un microcomputer. Per esempio, un'istruzione può complementare il contenuto del registro accumulatore della CPU — spostare i dati dell'accumulatore ad una parola di memoria — o mettere i dati in output attraverso una porta di I/O.

Per usare un microcomputer, tuttavia, dovete prima scegliere i dispositivi che vi daranno una sufficiente capacità logica; poi dovete progettare la logica per soddisfare le vostre necessità, creando una sequenza di istruzioni che, prese insieme, selezionano la capacità di logica del chip che soddisfano i bisogni della vostra applicazione. Una sequenza di istruzioni è un programma, e la programmazione è la reazione delle sequenze di istruzioni.

IL CONCETTO DI LINGUAGGIO DI PROGRAMMAZIONE

Il concetto di programmazione di un minicomputer è stato presentato nel Capitolo 3, dove abbiamo descritto un programma di cinque istruzioni per eseguire una addizione binaria.

In questo capitolo discutiamo dei tipi di istruzioni di cui avrà bisogno un vero e proprio microcomputer, e di come sono realmente scritti i programmi. In realtà, bisogna parlare del modo in cui si scrivono i programmi prima di parlare dei tipi di istruzioni, dato che useremo la terminologia della programmazione per descrivere le istruzioni.

Non vi è nulla che vi impedisca di creare un programma sotto forma di sequenza di codici di istruzioni binarie, proprio come appariranno nella memoria, o nel registro istruzioni. Il programma di addizione descritto nel Capitolo 4 può essere scritto in cifre binarie o esadecimali, in questo modo:

Programma sotto forma di matrice binaria	Versione esadecimale del programma
10011100	9C
00001010	0A
00110000	30
01000000	40
10011100	9C
00001010	0A
00110001	31
10000000	80
01100000	60

Se state per scrivere il vostro programma direttamente come sequenza di cifre binarie, i rischi di mettere al posto sbagliato uno 0 o un 1 sono molto alti, e i rischi di evitare gli errori sono molto bassi. Questa è una sfortuna perchè non è sufficiente per un programma essere preciso al 99,99%. Se il programma non è assolutamente preciso, c'è sempre la recondita possibilità che l'errore si manifesti in un momento inopportuno, con conseguenze disastrose. E' questa necessità di perfezione che fa sì che i programmatori cerchino di realizzare un dispositivo grazie al quale gli errori siano difficili da fare e facili da individuare.

In confronto alla creazione di un programma come sequenza di cifre binarie, il primo e più ovvio perfezionamento sarebbe codificare il programma usando le cifre esadecimali, e poi trovare un modo automatico di convertire le cifre esadecimali nel loro equivalente binario.

Il fatto di scrivere il programma in cifre esadecimali rende più difficile fare errori, perchè vi è una cifra esadecimale ogni quattro cifre binarie. Basandosi sulla teoria che ogni cifra offre una uguale probabilità di essere scritta sbagliata, è probabile che la programmazione in cifre esadecimali generi un quarto del numero di errori rispetto alle cifre binarie, perchè vi è un quarto delle cifre.

La programmazione in cifre esadecimali rende anche gli errori più facili da individuare, dato che, rilevando una cifra esadecimale posta nel modo sbagliato, anche se non è la cosa più semplice del mondo, avete la possibilità di individuare un 1 o uno 0 sbagliati in una configurazione binaria "ipnotizzante". Sono riprodotti di seguito i programmi binari ed esadecimali. Ognuno di essi ha un errore: provate a controllare quanto tempo impiegate a trovare gli errori:

Programma sotto forma di matrice binaria	Versione esadecimale del programma
10011100	9C
00001010	0A
00110000	30
01000000	40
10011100	9C
00001010	A0
00011001	31
10000000	80
01100000	60

Alla fine, comunque, il programma deve essere convertito in una sequenza binaria, perchè è questo il modo in cui verrà memorizzato — ed è questo il modo in cui ogni istruzione verrà interpretata nel registro istruzioni.

PROGRAMMI SORGENTE

Una telescrivente, o qualunque altro terminale con la tastiera appropriata, genera codici in carattere ASCII in risposta alla pressione sui tasti: perciò, supponiamo che un programma scritto in cifre esadecimali venga inizialmente generato come sequenza di codici in carattere ASCII.

Le cifre esadecimali sono rappresentate dalle cifre da 0 a 9 più le lettere da A ad F. I codici ASCII per queste cifre sono presi dall'Appendice A.

Supponete di scrivere il programma di addizione binaria su di un pezzo di carta, usando cifre esadecimali, com'è illustrato nella Figura 6-1. Questo è un programma sorgente.

Cifra esadecimale	Codice ASCII
0	00110000
1	00110001
2	00110010
3	00110011
4	00110100
5	00110101
6	00110110
7	00110111
8	00111000
9	00111001
A	01000001
B	01000010
C	01000011
D	01000100
E	01000101
F	01000110

Questo programma sorgente deve essere convertito in una forma che può essere caricata in memoria ed eseguita. Un modo per farlo è usare un nastro di carta, o banda perforata.

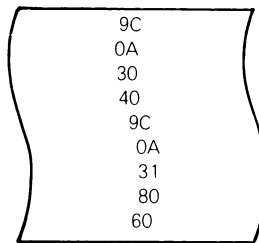
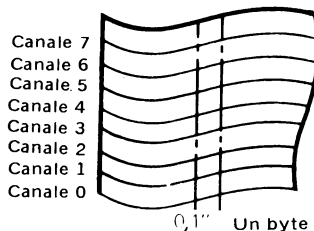



Figura 6-1. Programma sorgente scritto su carta

NASTRO DI CARTA

Un nastro di carta ha otto "canali" che rappresentano le otto cifre binarie di un byte. Un buco perforato in un qualunque canale rappresenta un 1, mentre l'assenza di un buco rappresenta uno 0. Dieci byte stanno in un pollice di nastro di carta. In altre parole, ogni 0,1" del nastro rappresenta un byte, in questo modo:

Di solito fra i canali 2 e 3 c'è una linea di perforazioni; le perforazioni vengono usate da una ruota dentata per far avanzare il nastro.

PROGRAMMI OGGETTO

Il nostro scopo è quello di convertire il programma sorgente, illustrato nella Figura 6-1, in un nastro di carta, come illustra la Figura 6-2. Il nastro di carta della Figura 6-2 è un'esatta rappresentazione dei codici di istruzioni binarie che verranno immagazzinati in memoria; le cifre 1 sono rappresentate da buchi, le cifre 0 sono rappresentate dalla mancanza di buchi. **Il programma illustrato nella Figura 6-2 è chiamato programma oggetto.**

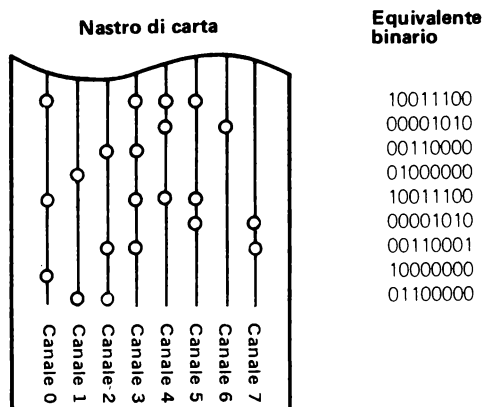



Figura 6-2. Programma oggetto su nastro di carta

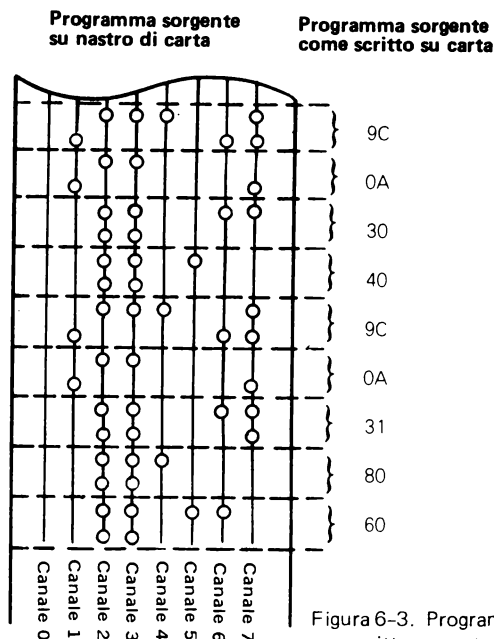



Figura 6-3. Programma sorgente scritto su nastro di carta

CREAZIONE DEI PROGRAMMI OGGETTO

La conversione del programma sorgente della Figura 6-1 nel programma oggetto della Figura 6-2 è un processo in due passi.

Dapprima le cifre esadecimali illustrate nella Figura 6-1, vengono battute alla tastiera. Supporremo che sia la tastiera di una telescrivente. Ogni cifra diventa un codice ASCII sul nastro di carta, come mostra la Figura 6-3.

PROGRAMMI DI EDITOR

Potreste creare il nastro di carta illustrato nella Figura 6-3 semplicemente attivando la perforatrice di una telescrivente, e poi premendo i tasti appropriati.

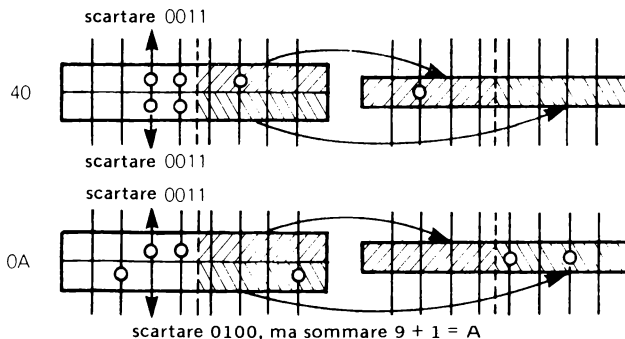
Con un po' più di fantasia, potreste collegare la telescrivente ad un computer, che esegua un programma per leggere i dati della tastiera e perforare il nastro di carta. Questo programma si chiama EDITOR.

Usare un programma di EDITOR per creare programmi sorgenti è una buona idea. Per esempio, il programma di Editor potrebbe essere scritto in modo da ignorare qualunque tasto che non sia una cifra esadecimale valida (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F). Dato che una telescrivente può anche leggere altrettanto bene le bande di carta di programmi sorgenti, apportarvi delle correzioni, poi perforare la versione corretta del programma sorgente. In questo modo, risparmiate il tempo di ribattere alla tastiera le parti del programma sorgente senza errori.

Dopo aver usato un Editor per creare un programma sorgente su nastro di carta, come mostra la Figura 6-3, eseguirete un altro programma che legge automaticamente il programma sorgente e crea un programma oggetto equivalente; per il momento chiameremo questo programma un programma CONVERTER.

Facendo riferimento alle Figure 6-2 e 6-3, la logica del programma Converter è molto semplice, ed è la seguente:

1) Combinare i quattro bit più a destra (di ordine inferiore) di ogni coppia di byte del programma sorgente in un byte del programma oggetto.
2) Se i canali da 0 a 3 del programma sorgente contengono 0011, scartate questi quattro bit ed usate i canali da 4 a 7 così come sono.
3) Se i canali da 0 a 3 del programma sorgente contengono 0100, scartate questi quattro bit ed usate 9 più il contenuto dei quattro bit dei canali da 4 a 7.

Questi tre passi logici possono essere illustrati nel modo seguente:

La banda di carta del programma oggetto, come viene creata dal programma Converter, può essere caricata direttamente nella memoria per essere eseguita. Il Capitolo 8 descrive come ciò avviene per i microcomputer.

MEZZI PER MEMORIZZARE I PROGRAMMI

Non dovete usare bande di carta come mezzo per creare programmi sorgente e programmi oggetto; infatti, il nastro di carta viene usato solo dai microcomputer più semplici. **Di solito si usa un mezzo magnetico, come un'unità disco, per memorizzare i programmi sorgenti ed oggetto (o qualunque altro dato).**

LINGUAGGIO ASSEMBLATORE

Perchè le cifre esadecimali sono più efficienti delle cifre binarie come mezzo di programmazione? Perchè le cifre esadecimali rendono più facile il compito del programmatore, lasciando al computer il compito più difficile.

Il fatto di semplificare il compito del programmatore — rendendo gli errori più difficili da fare e più facili da individuare — aumenta notevolmente il rendimento.
Il fatto di far convertire al computer un programma sorgente esadecimale in un programma oggetto binario — eseguendo un programma Converter — ci fa perdere poco tempo, perchè il programma Converter viene eseguito in secondi, o al massimo, in pochi minuti.

Spingiamo questo ragionamento un po' più avanti. **Anzichè programmare in cifre esadecimali, potremmo usare un linguaggio di programmazione che è ancora più semplice da capire per la persona che programma.**

Il linguaggio di programmazione sorgente sarà molto diverso da un programma oggetto in cifre binarie, perciò il programma Converter, che converte il programma in linguaggio sorgente in un programma oggetto binario, diventa più complesso; ma ciò rappresenta una penalità insignificante.

Quello che un linguaggio di programmazione tenta di fare è eliminare gli errori sintattici di programmazione — la cifra messa al posto sbagliato, il codice istruzione sbagliato — lasciando alla responsabilità del programmatore soltanto gli errori di logica, specifici dell'applicazione.

Il linguaggio assemblatore è il primo passo verso forme di programmazione più facilmente comprensibili per la persona che programma.

SINTASSI DEL LINGUAGGIO ASSEMBLATORE

Il linguaggio assemblatore di un mini o di un microcomputer consiste di un set di istruzioni, ognuna delle quali occupa una riga di programma sorgente. Ogni riga può essere divisa in quattro parti, o campi, così:

Label	Campo codice mnemonico	Operando	Commento
	LIM	DC0,ADDR1	; Carica l'indirizzo sorgente nel DC
HERE	LMA		; Carica il dato nell'Accumulatore
	AIA	H'0F'	; Maschera i quattro bit più significativi
	BZ	OUT	; Salta se il risultato è 0
	SRA		; Memorizza i dati mascherati
	INC	DC0	; Incrementa il Data Counter
	JMP	HERE	; Ritorna per il byte successivo
OUT			; Istruzione successiva

CAMPO CODICE MNEMONICO

Ogni istruzione del programma sorgente rappresenta una istruzione del programma oggetto.
Consideriamo prima il campo codice menmonico, che potrebbe essere evidenziato in questo modo:

Label	Campo codice mnemonico	Operando	Commento
	LIM	DC0,ADDR1	; Carica l'indirizzo sorgente nel DC
HERE	LMA		; Carica il dato nell'Accumulatore
	AIA	H'0F'	; Maschera i quattro bit più significativi
	BZ	OUT	; Salta se il risultato è 0
	SRA		; Memorizza i dati mascherati
	INC	DC0	; Incrementa il Data Counter
	JMP	HERE	; Ritorna per il byte successivo
OUT			; Istruzione successiva

Il campo codice menmonico è il campo più importante in un'istruzione in linguaggio assemblatore, ed è l'unico campo che deve sempre contenere qualcosa. Esso contiene un gruppo di lettere che costituiscono un codice che identifica l'istruzione del programma sorgente.

PROGRAMMA ASSEMBLATORE (ASSEMBLER)

Il programma di conversione usato per convertire il programma sorgente in linguaggio assemblatore in programma oggetto binario, si chiama ASSEMBLER. L'Assembler legge il codice menmonico, come un gruppo di caratteri ASCII, e sostituisce il codice istruzione binario per generare un programma oggetto.

Consideriamo l'istruzione specificata dal campo codice menmonico SRA. Questa istruzione esegue la stessa operazione dell'istruzione 5 del programma di addizione binaria descritto nel Capitolo 4, dove viene mostrato usando il codice istruzioni 60_{16}. L'Assembler deve quindi avere una logica che generi il codice istruzione 60_{16} nell'incontrare SRA nel campo codice mnemonico di un'istruzione del programma sorgente:

Programma sorgente	Forma binaria del campo codice mnemonico visto dall'Assembler	Forma binaria del programma oggetto creato dall'Assembler
S	01010011	
R	01010010 → Logica dell'Assembler	00110000
A	01000001	

Notate attentamente che solo i codici istruzione binari di un microcomputer, cioè i codici oggetto, sono sacri e inalterabili. I campi codice mnemonico del programma sorgente sono scelti arbitrariamente e possono essere cambiati in qualunque momento semplicemente riscrivendo l'Assembler per riconoscere il nuovo campo codice mnemonico del programma sorgente.

Tutti i manuali di programmazione dei microcomputer definiscono le istruzioni usando campi codice mnemonico dei programmi sorgenti. Il fatto che la selezione dei campi codice mnemonico sia un fatto molto arbitrario è dimostrato dal fatto che solo raramente due microcomputer diversi usano lo stesso campo mnemonico per identificare codici istruzione che fanno la stessa cosa. Infatti, la selezione dei campi codice mnemonico delle istruzioni è un fatto di pura scelta personale del progettista.
Molti utenti dell'Intel 8080, ad esempio, hanno creato set di istruzioni per proprio conto ed hanno già risolto il problema di scrivere i loro programmi Assembler per riconoscere i loro nuovi campi codice mnemonico.

Vediamo come si usano più campi codice mnemonico per rappresentare la stessa istruzione.

Abbiamo appena mostrato come il campo codice mnemonico SRA venga convertito nel codice istruzione del programma oggetto 60_{16}. Si potrebbe scrivere un altro programma Assembler per convertire il campo codice mnemonico XYZ nel codice istruzione del programma oggetto 60_{16}. Si potrebbe poi scrivere un terzo programma Assembler per convertire o SRA o XYZ in 60_{16}.

I campi codice mnemonico usati in questo capitolo sono stati scelti fra quelli più semplici usati dai vari microcomputer.
I campi codice mnemonico LIM, LMA, AIA, BZ, SRA, INC e JMP provengono dallo ipotetico set di istruzioni del microcomputer creato nel Capitolo 7.

Parleremo ora del campo etichetta (label), evidenziato in questo modo:

Label	Campo codice mnemonico	Operando	Commento
	LIM	DC0,ADDR1	; Carica l'indirizzo sorgente nel DC
HERE	LMA		; Carica il dato nell'Accumulatore
	AIA	H'0F'	; Maschera i quattro bit più significativi
	BZ	OUT	; Salta se il risultato è 0
	SRA		; Memorizza i dati mascherati
	INC	DC0	; Incrementa il Data Counter
	JMP	HERE	; Ritorna per il byte successivo
OUT			; Istruzione successiva

CAMPO ETICHETTA

Il campo etichetta può contenere o meno qualcosa. Se c'è qualcosa nel campo etichetta, è possibile indirizzare l'istruzione. In altre parole, non si identifica un'istruzione tramite la sua posizione nella memoria di programma (come avveniva nel Capitolo 4), perchè nel momento in cui si scrive il programma, si può non sapere in che punto della memoria finirà l'istruzione. In questo caso si dà all'istruzione un nome, o un'etichetta.
Facciamo riferimento all'esempio precedente. L'istruzione etichetta HERE deve essere identificata, perchè più avanti c'è un'istruzione che specifica un cambiamento della sequenza dell'esecuzione. L'istruzione:

```
JMP        HERE        ; Ritorno per il byte successivo
```

specifica che l'istruzione etichettata HERE è la prossima istruzione che verrà eseguita. Questa è un'istruzione di salto; può essere usata per illustrare il significa di un'etichetta rappresentando in questo modo la sequenza dell'esecuzione del programma:

L'Assembler dovrà tenere conto del punto di memoria in cui finiranno le istruzioni, perchè esso dovrà sostituire tutte le label con un indirizzo di memoria reale.

Supponiamo che il programma oggetto del suddetto programma sorgente in linguaggio assemblatore vada ad occupare la memoria in questo modo:

Locazione della memoria del programma oggetto	Label	Campo codice mnemonico	Operando	Commento
03FF,0400,0401		LIM	DC0,ADDR1	; Carica l'indirizzo ; sorgente nel DC
0402	HERE	LMA		; Carica il dato nello ; Accumulatore
0403,0404		AIA	H'0F'	; Maschera i quattro ; bit più significativi
0405,0406		BZ	OUT	; Salta se il risultato ; è 0
0407		SRA		; Memorizza i dati ; mascherati
0408		INC	DC0	; Incrementa il Data ; Counter
0409,040A		JMP	HERE	; Ritorna per il byte ; successivo
040B	OUT			; Istruzione succes- ; siva

L'Assembler assegnerà il valore 0402 a HERE, e 0408 ad OUT.
Il codice istruzione binario per l'istruzione JMP potrebbe essere BC_{16}. Se la label HERE ha il valore di 0402_{16}, l'Assembler convertirà la istruzione del programma sorgente:

```
JMP        HERE
```

nei tre byte di programma oggetto:

```
BC
04
02
```

Se voi spostate il programma, in modo che il codice oggetto per:

```
HERE       LMA
```

occupasse ora il byte della memoria di programma con indirizzo $0C7A_{16}$, l'Assembler convertirebbe:

```
JMP        HERE
```

nei tre byte di programma oggetto:

```
BC
0C
7A
```

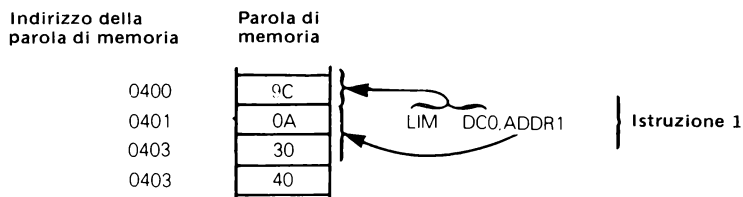
CAMPO OPERANDO

Il campo operando può essere evidenziato in questo modo:

Label	Campo codice mnemonico	Operando	Commento
	LIM	DC0,ADDR1	; Carica l'indirizzo sorgente nel DC
HERE	LMA		; Carica il dato nell'Accumulatore
	AIA	H'0F'	; Maschera i quattro bit più significativi
	BZ	OUT	; Salta se il risultato è 0
	SRA		; Memorizza i dati mascherati
	INC	DC0	; Incrementa il Data Counter
	JMP	HERE	; Ritorna per il byte successivo
OUT			; Istruzione successiva

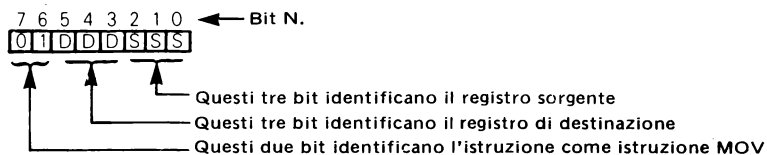
H'0F' significa $0F_{16}$. Non potete usare degli indici di programma sorgente, così bisogna scegliere un'alternativa ragionevole.
Di solito, ma non sempre, il campo operando fornisce informazioni che l'Assembler userà per creare il secondo byte o il secondo e il terzo byte di un'istruzione che richiede più di un byte di codice oggetto. Per esempio, supponiamo che l'istruzione del programma sorgente

LIM DC0, ADDR1

sia l'istruzione 1 del programma di addizione binaria illustrato nel Capitolo 3. L'Assembler interpreterà l'istruzione del programma sorgente in questo modo:

Non vi sono regole che impongano che l'operando debba specificare o possa solo specificare il secondo e il terzo byte del codice istruzioni del programma oggetto. Il microcomputer Intel 8080, per esempio, ha sette registri accumulatore ed una sola istruzione che sposta i dati da un registro ad un altro. Questa istruzione del programma sorgente è così scritta:

MOV D,S

dove D specifica il registro di destinazione,
S specifica il registro sorgente
e D,S costituisce il campo operando.

E l'istruzione MOV crea quest'unico byte del programma oggetto.

Ora guardiamo il contenuto dei campi operando nel programma che illustriamo in questo capitolo:

DC0, ADDR1 DC0 identifica il Data Counter nel quale dei dati immediati devono essere caricati. ADDR1 è una label che rappresenta l'indirizzo che deve essere caricato nel Data Counter. L'Assembler convertirà ADDR1 in un valore binario di 16 bit.

H'0F' specifica il valore immediato $0F_{16}$ di due cifre esadecimali. Il campo mnemonico AIA dell'istruzione sta per AND Immediate. Questa combinazione di codice mnemonico e di operando fa sì che qualunque cosa sia contenuta nell'accumulatore venga messa in AND col valore reale nel campo operando. In questo caso, dato che il campo operando è $0F_{16}$, ha l'effetto di posizionare a zero i quattro bit di

ordine superiore dell'accumulatore lasciando invece inalterati i quattro bit di ordine inferiore dell'accumulatore.

OUT e HERE — che appaiono nei campi operando, identificano le istruzioni label OUT e HERE. L'istruzione:

```
BZ          OUT          ; Esci se il risultato è 0
```

specifica che, se l'accumulatore contiene un valore zero in seguito all'istruzione di AND Immediate, l'istruzione seguente da eseguire deve essere l'istruzione con label OUT, non l'istruzione SRA che, nella normale sequenza, verrebbe eseguita subito dopo. L'istruzione:

```
JMP         HERE         ; Ritorno per il byte successivo
```

stabilisce che l'istruzione da eseguire successivamente deve essere incondizionatamente l'istruzione con la label HERE, non l'istruzione con la label OUT, che, venendo subito dopo, nella sequenza, avrebbe dovuto essere l'istruzione da eseguire successivamente.

DC0 — specifica il Data Counter il cui contenuto deve essere incrementato.

CAMPO COMMENTO

Il campo commento contiene delle informazioni che rendono il programma più facile da leggere ma che non hanno effetto sul programma oggetto binario creato dall'Assembler. In altre parole, l'Assembler ignora il campo commento.

CAMPO DI IDENTIFICAZIONE

In che modo l'Assembler potrà sapere dove finisce un campo e ne incomincia un altro? Di solito si usano degli spazi per separare i campi, e l'Assembler usa queste regole:

1) Tutti i caratteri, dal primo carattere di una riga fino al primo spazio incontrato, costituiscono il campo etichetta.
2) Gli spazi contigui sono trattati come se fossero un unico spazio.
3) Tutti i caratteri fra il primo e il secondo spazio (o spazi contigui) vengono interpretati come campo codice mnemonico.
4) Se il campo codice mnemonico non richiede un operando, l'Assembler smette qui, supponendo che tutto quello che segue sia commento.
5) Se il campo mnemonico richiede un operando, l'Assembler suppone che tutti i caratteri fra il secondo e il terzo spazio (o spazi contigui) costituiscano il campo operando.
6) Talvolta i campi commento sono preceduti da un carattere prefissato. In questo caso abbiamo usato il punto e virgola.

Gli spazi delimitatori dei campi possono essere illustrati in questo modo, in accordo con le regole suddette:

Label	**Campo codice mnemonico**	**Operando**	**Commento**
	LIM	DC0,ADDR1	; Carica l'indirizzo sorgente nel DC
HERE	LMA		; Carica il dato nell'Accumulatore
	AIA	H'0F'	; Maschera i quattro bit più significativi
	BZ	OUT	; Salta se il risultato è 0
	SRA		; Memorizza i dati mascherati
	INC	DC0	; Incrementa il Data Counter
	JMP	HERE	; Ritorna per il byte successivo
OUT			; Istruzione successiva

DIRETTIVE DI ASSEMBLER

Un programma in linguaggio assemblatore, come la sequenza di sette istruzioni che abbiamo usato per illustrare i campi delle istruzioni, non può essere assemblato così come si trova. Per dare alle label OUT e HERE dei valori binari fissi bisogna dire all'Assembler dove eventualmente risiederà il programma oggetto nella memoria di programma.

Vi è una classe di istruzioni, chiamata Direttive di Assembler, che userete per fornire all'Assembler le informazioni che non può dedurre da solo.

DIRETTIVE DI ORIGINE

Quando abbiamo spiegato in che modo le label OUT e HERE nel campo operando dovrebbero essere interpretate dall'Assembler, abbiamo illustrato una sequenza di programma che occupa le posizioni di memoria che iniziano a 03FF. Bisogna quindi specificare l'origine all'Assembler usando una direttiva di Assembler, in questo modo:

Label	Campo codice mnemonico	Operando	Commento
	ORG	H'03FF'	
	LIM	DC0,ADDR1	; Carica l'indirizzo sorgente nel DC
HERE	LMA		; Carica il dato nell'Accumulatore
	AIA	H'0F'	; Maschera i quattro bit più significativi
	BZ	OUT	; Salta se il risultato è 0
	SRA		; Memorizza i dati mascherati
	INC	DC0	; Incrementa il Data Counter
	JMP	HERE	; Ritorna per il byte successivo
OUT			; Istruzione successiva

La direttiva d'origine Assembler non genera nessun codice oggetto. Il suo unico scopo nel programma è quello di dire all'Assembler dove verrà posizionato il codice oggetto nella memoria di programma, e quindi come calcolare gli indirizzi di memoria binari reali che devono essere sostituiti alla label delle istruzioni.

DIRETTIVE DI FINE

L'origine è l'unica direttiva di Assembler che è assolutamente ingiuntiva. Un'altra direttiva Assembler che è sempre presente, perchè facilita il compito dell'Assembler, è la direttiva di fine. Essa è l'ultima istruzione del programma e dice all'Assembler che non vi sono più istruzioni eseguibili. La direttiva di Assembler END si può illustrare in questo modo:

Label	Campo codice mnemonico	Operando	Commento
	ORG	H'03FF'	
	LIM	DC0,ADDR1	; Carica l'indirizzo sorgente nel DC
HERE	LMA		; Carica il dato nell'Accumulatore
	AIA	H'0F'	; Maschera i quattro bit più significativi
	BZ	OUT	; Salta se il risultato è 0
	SRA		; Memorizza i dati mascherati
	INC	DC0	; Incrementa il Data Counter
	JMP	HERE	; Ritorna per il byte successivo
	END		

DIRETTIVA DI EQUALIZZAZIONE

La direttiva Assembler di equalizzazione (o equate) è un'altra molto spesso presente, perchè facilita notevolmente la programmazione in linguaggio assemblatore. Si usa la direttiva Assembler di equalizzazione per assegnare un valore ad un sim-

bolo o ad una label. Consideriamo l'istruzione:

```
          AIA         H"OF          ; Maschera i 4 bit più significativi
```

L'operando H'OF' potrebbe essere sostituito da un simbolo che è uguagliato al valore $0F_{16}$. Lo illustriamo così:

Label	Campo codice mnemonico	Operando	Commento
VALUE	EQU	H'0F'	
	ORG	H'03FF'	
	LIM	DC0,ADDR1	; Carica l'indirizzo sorgente nel DC
HERE	LMA		; Carica il dato nell'Accumulatore
	AIA	VALUE	; Maschera i quattro bit più significativi
	BZ	OUT	; Salta se il risultato è 0
	SRA		; Memorizza i dati mascherati
	INC	DC0	; Incrementa il Data Counter
	JMP	HERE	; Ritorna per il byte successivo
OUT			; Istruzione successiva

Una direttiva Assembler di equalizzazione potrebbe essere usata anche per assegnare un valore all'indirizzo etichettato ADDR1. Comunque dovreste farlo solo se ADDR1 non esistesse come etichetta di una istruzione in qualche altra parte del programma.

DEFINIZIONE DI COSTANTE

Vi sono due campi codice mnemonico che appaiono in tutti i linguaggi assemblatori e non sono né istruzioni né direttive di assembler. Essi sono la definizione di costante e la definizione di indirizzo.

DEFINIZIONE DI INDIRIZZO

Il campo codice mnemonico definizione di costante si usa per specificare un solo byte di dati. Il campo codice mnemonico definizione di indirizzo si usa per specificare due byte di dati. Ecco un esempio di come verrebbero usati questi due campi codice mnemonico. L'istruzione che segue:

```
             ORG     H'0700'
             DC      H'3A'
    ADDR1    DA      H'27AC'
    VALUE    DC      H'0F'
```

farebbe sì che l'Assembler crei direttamente la seguente mappa di memoria:

0700	3A
0701	27
0702	AC
0703	0F
0704	
0705	
0706	

INDIRIZZAMENTO DELLA MEMORIA

Dell'indirizzamento della memoria abbiamo già parlato nel Capitolo 4, dove era necessario parlare un po' di quest'argomento allo scopo di definire i registri di cui una CPU ha bisogno. Tratteremo ora fino in fondo questo argomento prima di arrivare a definire il set di istruzioni di un microcomputer.

L'INDIRIZZAMENTO DI MEMORIA DEI MICROCOMPUTER— DA COSA NASCE

Inizieremo a parlare dei modi di indirizzamento nei microcomputer esaminando l'argomento da un punto di vista complessivo.

I precursori di tutti i set di istruzioni del microcomputer sono quelli dell'Intel 8080 e del microcomputer Datapoint 2200. Questi due dispositivi hanno lo stesso set di istruzioni, ma il Datapoint 2200 esegue le istruzioni circa dieci volte più velocemente. Mr. Vic Poor (e soci) hanno sviluppato il set di istruzioni limitatamente all'ambiente di elaborazione dati dei "terminali intelligenti". Quello che avevano in mente non era la sostituzione della logica discreta. Il set di istruzioni di Vic Poor era deliberatamente limitato, per adattarsi ai confini della tecnologia dell'integrazione su larga scala (LSI) al punto in cui essa era nel 1969-1970. **Le capacità di indirizzamento di memoria del set di istruzioni era al di là delle necessità e quindi non desiderato.**

L'Intel che sviluppò inizialmente il microcomputer 8008 su richiesta della Datapoint, trovò un buon mercato per il prodotto nella sostituzione della logica discreta — un mercato per il quale il set di istruzioni non era mai stato concepito.

I set di istruzioni dei microcomputer che seguirono si sono evoluti come risultato di due influenze contrapposte.

a) I progettisti di microcomputer hanno incorporato le caratteristiche dei minicom puter nella misura in cui lo permettevano i progressi nella tecnologia LSI — mentre non era definibile nessuna base di utenti di microcomputer.

b) Ora che tale base incomincia ad emergere, i progettisti di microcomputer rispondono direttamente ai bisogni dell'utente.

Le suddette influenze (a) e (b) sono in effetti diverse. I bisogni degli utenti di microcomputer non sempre sono pienamente soddisfatti dai set di istruzioni dei minicomputer, un fatto questo su cui torneremo continuamente in questo capitolo.

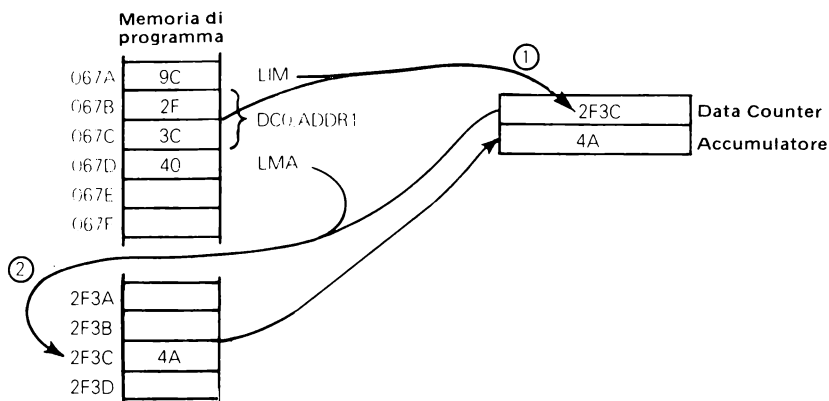
INDIRIZZAMENTO DI MEMORIA IMPLICITO

Un'istruzione che usa l'indirizzamento di memoria implicito specifica il contenuto del Data Counter come indirizzo di memoria.

L'indirizzamento di memoria implicito è stato descritto dettagliatamente nel Capitolo 4; perciò, riassumeremo semplicemente questo metodo d'indirizzamento.
L'uso del Data Counter per indirizzare memoria è un processo in due fasi:

- Primo, l'indirizzo di memoria richiesto deve essere caricato nel Data Counter.
- Poi, viene eseguita un'istruzione di riferimento alla memoria ad un solo byte, dove il Data Counter contiene l'indirizzo della posizione di memoria cui bisogna far riferimento.

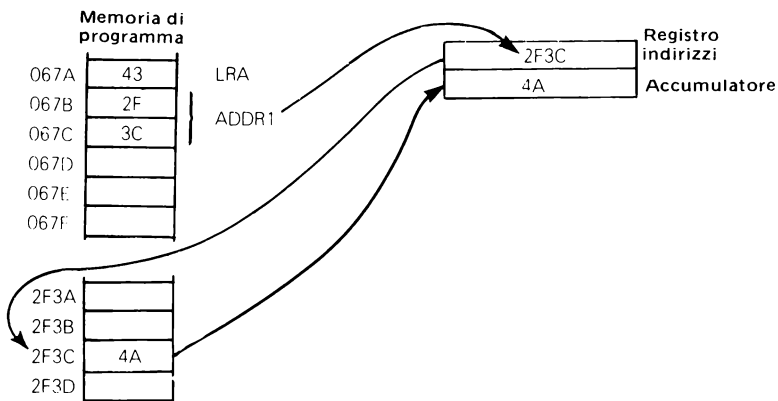
Consideriamo le prime due istruzioni dell'esempio di programmazione che abbiamo usato in questo capitolo. L'esecuzione delle due istruzioni può essere illustrata come si può vedere dalla figura a pagina seguente.

Il codice oggetto della prima istruzione (LIM DC0, ADDR1) occupa tre byte della memoria di programma, agli indirizzi 067A, 067B e 067C. Questi indirizzi di memoria sono stati selezionati arbitrariamente. Il byte 067A contiene un codice oggetto di 8 bit, che rappresenta il campo codice mnemonico dell'istruzione LIM DC0, ADDR1. Questa specifica, che il contenuto dei due byte seguenti della memoria di programma devono essere caricati, come un valore di 16 bit, nel Data Counter DC0. Ricordate

che il codice binario reale che appare nella parola di memoria 067A varia da microcomputer a microcomputer.

La seconda istruzione, con campo codice mnemonico LMA, specifica che il contenuto della posizione di memoria che è indirizzata dal Data Counter deve essere caricato nell'accumulatore. E' questo l'indirizzamento di memoria IMPLICITO, perchè l'istruzione di riferimento alla memoria, in questo caso LMA, non specifica un indirizzo di memoria; invece, decide che la posizione di memoria il cui indirizzo è contenuto nel Data Counter DC0 è la posizione di memoria cui si deve far riferimento.

INDIRIZZAMENTO DI MEMORIA DIRETTO

Un'istruzione con indirizzamento di memoria diretto specifica direttamente l'indirizzo della posizione di memoria cui si deve far riferimento.

Nel Capitolo 4 è stato descritto un indirizzamento di memoria diretto insieme all'indirizzamento di memoria implicito. Nel contesto della sequenza istruzioni che usiamo in questo capitolo, le istruzioni LIM e LMA potrebbero essere raggruppate in una istruzione di riferimento alla memoria diretta, in questo modo:

Un registro indirizzi esegue la stessa funzione di un Data Counter, ma lo fa in modo transitorio.

Un'istruzione di riferimento diretto alla memoria comincia sempre con un indirizzo di memoria che viene elaborato e caricato nel registro indirizzi. Questo diventa l'indirizzo della posizione di memoria cui si deve far riferimento.

L'indirizzamento diretto è il modo più semplice d'indirizzamento usato dai minicomputer. L'indirizzamento di memorie implicito è tipico dei microcomputer.

CONFRONTO TRA L'INDIRIZZAMENTO DIRETTO E QUELLO IMPLICITO

A proposito del confronto diretto, il registro indirizzi di un minicomputer viene chiamato anche registro non programmabile. Ciò significa che un minicomputer non ha istruzioni che caricano semplicemente dei dati nel registro indirizzi o ne modificano il contenuto. Il processo di cambiamento del contenuto del registro indirizzi è sempre una fase transitoria nel corso dell'esecuzione di un'istruzione di riferimento alla memoria.

Il Data Counter di un microcomputer è programmabile. Infatti, tutti i microcomputer hanno un certo numero di istruzioni che caricano semplicemente dei dati nel Data Counter, o ne modificano il contenuto e non fanno nient'altro.

Alcuni microcomputer hanno sia l'indirizzamento di memoria implicito che quello diretto. Questi microcomputer hanno uno o più Data Counter per l'indirizzamento di memoria implicito, più un registro indirizzi addizionale per l'indirizzamento diretto di memoria.

I primi microcomputer usavano solo l'indirizzamento in memoria implicito, perchè era facile da progettare nell'unità di controllo della CPU; non vi erano altre ragioni. Lo svantaggio dell'uso dell'indirizzamento in memoria implicito sta nel fatto che esso occupa due istruzioni per fare quello che potrebbe fare una sola istruzione d'indirizzamento diretto. La tecnologia LSI è arrivata al punto in cui i progettisti di microcomputer potrebbero eliminare l'indirizzamento implicito, ma non lo hanno fatto. Oggi la maggior parte dei microcomputer ha un numero limitato di istruzioni con indirizzamento diretto, ma l'indirizzamento implicito rimane quello standard: perchè? Perchè alcune variazioni necessarie dell'indirizzamento diretto generano caratteristiche assolutamente indesiderabili quando i programmi vengono memorizzati nella ROM.

VARIAZIONI DELL'INDIRIZZAMENTO DI MEMORIA DIRETTO

Considereremo dapprima le variazioni dell'indirizzamento diretto come vengono applicate al minicomputer con parole di 12 e 16 bit. Questo è un buon inizio, dato che le variazioni dell'indirizzamento diretto si sono evolute sotto forma di caratteristiche dell'indirizzamento dei minicomputer.

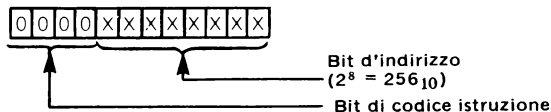
Una parola di 16 bit permette ad un minicomputer di avere 65.536 istruzioni diverse nel suo set di istruzioni. Una parola di 12 bit permette di avere 4.096 istruzioni diverse nel set di istruzioni. Questi numeri sono ridicolarmente alti. Perciò i minicomputer dividono le parole delle istruzioni in bit di codice istruzione e bit d'indirizzo.

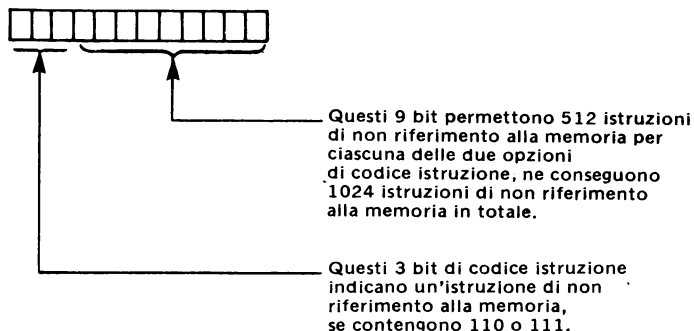
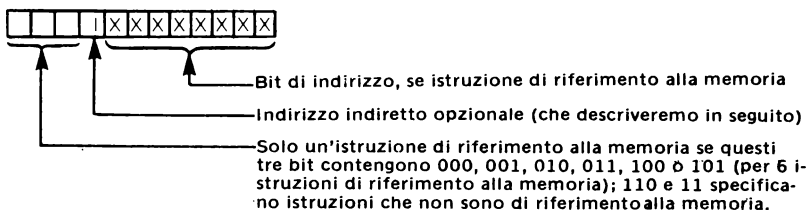
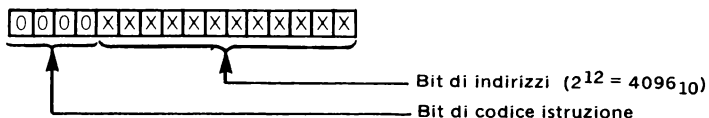
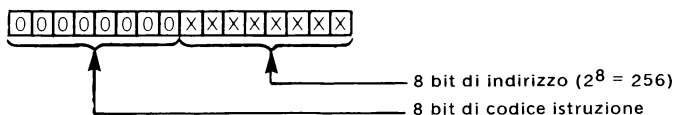
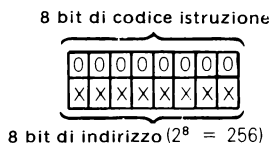
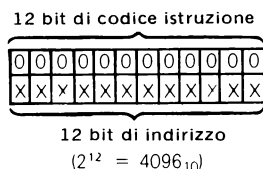
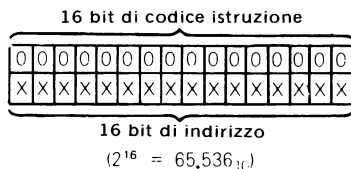
INDIRIZZAMENTO DIRETTO CON PAROLE DI 12 BIT

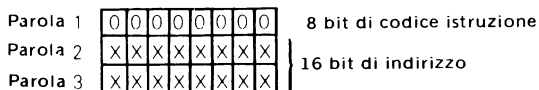
PDP – 8

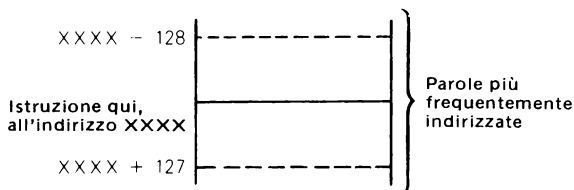
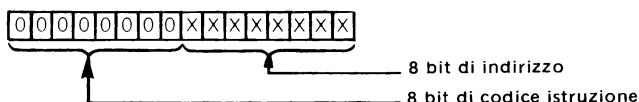
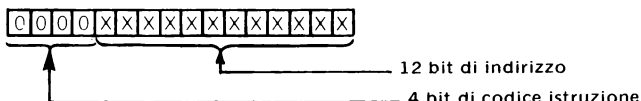
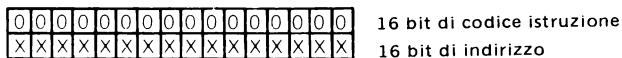
INTERSIL IM6100

Consideriamo dapprima un minicomputer con una parola di 12 bit. Il PDP-8 della Digital Equipment Corporation, il primo minicomputer famoso nel mondo, usa una parola di 12 bit. La CPU del PDP-8 è ora costruita dalla Intersil su di un singolo chip, chiamato

IM6100. La parola di 12 bit può essere usata in questo modo:

Otto bit d'indirizzo permettono a questa istruzione di indirizzare direttamente fino a 256 parole di memoria; 256 parole costituiscono una memoria molto piccola, perciò dobbiamo cercare un modo di espandere il nostro range d'indirizzamento in memoria, senza usare altri bit.

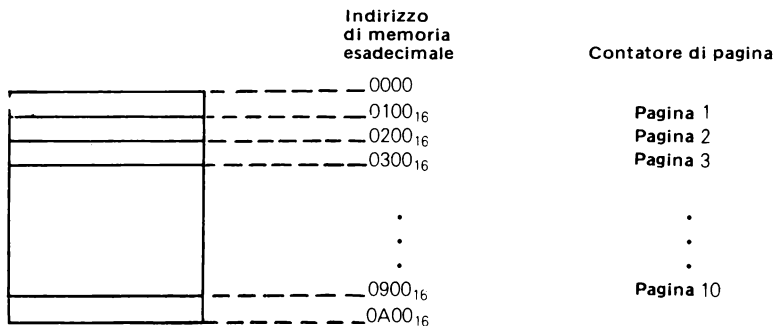
Saranno sufficienti quattro bit di codice istruzione? Decisamente sì. Ricordate, la separazione di dodici bit in quattro bit di codice istruzione e otto bit d'indirizzo si applica alle istruzioni di riferimento alla memoria: Ecco come il PDP-8 e IM6100 interpretano le istruzioni di riferimento alla memoria.

Un computer di 16 bit potrebbe indirizzare 4096_{10} parole di memoria con 12 bit di un codice istruzione:

Il computer di 16 bit potrebbe offrire più possibilità di codici istruzione, ed un range d'indirizzamento minore, se organizzato come segue:

La maggior parte dei minicomputer dividono le parole delle loro istruzioni in otto bit d'indirizzo e otto bit di codice istruzione, come precedentemente illustrato. Un microcomputer di 8 bit può facilmente raggiungere lo stesso risultato, usando due parole di 8 bit, in questo modo:

Un computer di 12 bit potrebbe indirizzare 4096_{10} parole fornendo due parole per ogni istruzione:

Sebbene sia possibile usare due parole di 12 bit per ogni istruzione, tutte le istruzioni del PDP-8 sono, in realtà istruzioni di una sola parola.
Se un computer di 16 bit usa due parole per ogni istruzione, esso può indirizzare direttamente 65.536_{10} parole di memoria:

Gli esempi di programmazione per microcomputer che abbiamo usato in precedenza in questo libro usano indirizzi di memoria di 16 bit per mezzo di tre parole a 8 bit, in questo modo:

Parola 1 | 0 0 0 0 0 0 0 0 | 8 bit di codice istruzione
Parola 2 | X X X X X X X X | 16 bit di indirizzo
Parola 3 | X X X X X X X X |

I BIT D'INDIRIZZO — IL NUMERO OTTIMALE

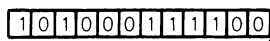
E' chiaro che, in un modo o nell'altro, le istruzioni possono avere da 8 a 16 bit. Qual'è il numero ideale? Nelle applicazioni dei minicomputer, secondo le statistiche troviamo che l'80-90% delle istruzioni di salto hanno bisogno solo di un range d'indirizzamento di ± 128 parole (indirizzabili con otto bit d'indirizzo):

La maggior parte dei minicomputer forniscono più di un formato della parola di istruzione, compresi due o tutti e tre i seguenti:

Le istruzioni con un range d'indirizzamento ristretto si chiamano istruzioni SHORT FORM hanno bit d'indirizzo sufficienti per indirizzare direttamente tutte le parole di memoria.

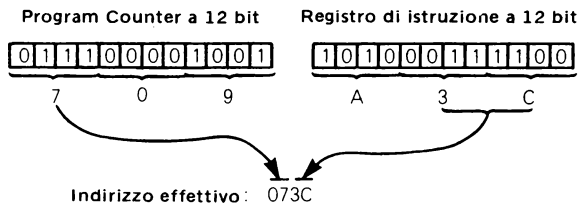
INDIRIZZAMENTO DIRETTO CON PAGINE

Tutti i computer hanno delle istruzioni con un range d'indirizzamento limitato. Se tutte le istruzioni di un computer sono soggette ad un range d'indirizzamento limitato, si dice che il computer è relativo a pagine o impaginato (PAGED).

Per illustrare il paging, consideriamo un minicomputer di 12 bit con otto bit d'indirizzo per ogni istruzione. La memoria è effettivamente segmentata in 256_{10} (100_{16}) pagine di parole.

Gli otto bit d'indirizzo di un'istruzione forniscono le due cifre esadecimali di ordine inferiore di un indirizzo di memoria a quattro cifre; le due cifre di ordine superiore vengono prese dal Program Counter. Perciò l'istruzione

JMP H'3C'

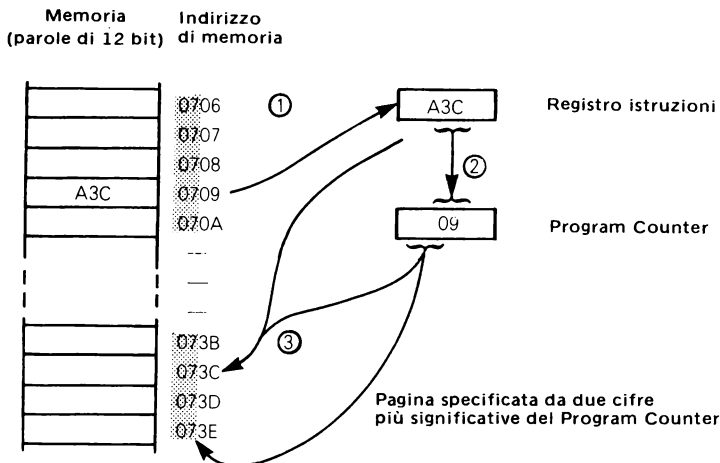
verrebbe codificata in una parola di 12 bit, così:

INDIRIZZO DI MEMORIA EFFETTIVO

Supponiamo che l'istruzione di salto sia memorizzata nella parola di memoria di indirizzo 0709_{16}. Dopo che l'istruzione di salto è stata prelevata dalla memoria, il Program Counter conterrà il valore $070A_{16}$. L'indirizzo di memoria effettivo, quindi, è dato dalle due cifre di ordine superiore del Program Counter, più le due cifre di ordine inferiore dell'istruzione, in questo modo:

NUMERO DI PAGINA

Le cifre di ordine superiore, prese dal Program Counter, sono chiamate numero di pagina. Le cifre di ordine inferiore, fornite dal codice istruzione, sono l'indirizzo all'interno della pagina. Combinando le due parti dell'indirizzo, si ottiene l'indirizzo di memoria effettivo. Il termine "indirizzo di memoria effettivo" viene applicato a tutti gli indirizzi di memoria che devono essere calcolati in qualche modo usando informazioni fornite dall'istruzione.

Come illustrato di seguito, l'istruzione

JMP H'3C'

farà sì che il valore $073C_{16}$ venga caricato nel Program Counter, così l'istruzione seguente verrà prelevata dalla posizione di memoria $073C_{16}$.

L'illustrazione viene descritta in questo modo, usando i riferimenti (1), (2) e (3)

(1) Il Program Counter indirizza la parola di memoria 0709_{16}. Il contenuto di questa parola di memoria, $A3C_{16}$, viene prelevato dalla memoria e memorizzato nel registro istruzioni. Il Program Counter viene incrementato a $070A_{16}$; così, esso indirizza la parola di memoria di programma successiva.

(2) Il codice istruzione nel registro istruzioni è un salto incondizionato. Le due cifre di ordine inferiore del registro istruzioni ($3C_{16}$) vengono spostate nelle due cifre di ordine inferiore del Program Counter, che contiene ora $073C_{16}$.

(3) L'istruzione seguente sarà prelevata dalla posizione di memoria $073C_{16}$.

Vediamo un altro esempio. L'istruzione:

LMA H'6C'

posizionata nella pagina $2F_{16}$ farebbe sì che il contenuto della posizione di memoria $2F6C_{16}$ venga caricato nell'accumulatore. La stessa istruzione a pagina $1C_{16}$ farebbe sì che il contenuto della posizione di memoria $1C6C_{16}$ venga caricato nell'accumulatore.

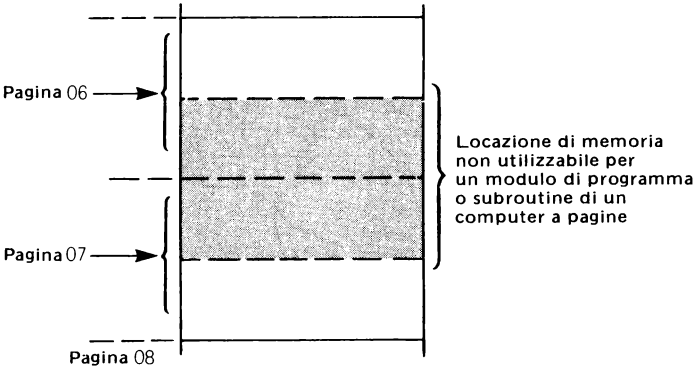
ERRORE DI CONFINE DI PAGINA

Con i computer con la memoria organizzata a pagine si verifica un errore dovuto ai confini delle pagine. Ricordate che il Program Counter viene incrementato dopo che viene prelevata dalla memoria una istruzione. Perciò il numero di pagine è determinato dalle cifre di ordine superiore del Program Counter, DOPO che il Program Counter è stato incrementato. Supponiamo che l'istruzione:

LMA H'6C'

sia posizionata all'indirizzo di memoria $2FFF_{16}$. Dopo che questa istruzione è stata prelevata dalla memoria, il Program Counter conterrà 3000_{16}. Ora il contenuto della

parola di memoria con indirizzo $306C_{16}$ verrebbe caricato nell'accumulatore, al posto della parola di memoria con indirizzo $2F6C_{16}$.

La restrizione più severa imposta dalle pagine fisse è che un'istruzione non può far riferimento a parole di memoria all'esterno della pagina in cui si trova l'istruzione stessa: nè per leggere dati, nè per scrivere dati, nè per eseguire un salto. Perciò**, i programmi non possono trovarsi sul confine di una pagina:**

L'impaginazione consuma la memoria del computer, perchè richiede che i programmi accedano ai dati sulla pagina o per mezzo di indirizzi memorizzati sulla pagina dove si trovano i dati. Perciò i numeri comunemente usati dal programma in molte pagine diverse devono essere ripetutamente memorizzati su ogni pagina, oppure dobbiamo aggiungere una nuova possibilità d'indirizzamento anche se strutturato a pagine.

Inoltre, quando scriviamo moduli di programma e subroutine, è difficile fare in modo che ogni modulo riempia esattamente una pagina. Come risultato, si consuma una piccola parte di memoria alla fine di ogni pagina, dato che questa parte è troppo piccola per andar bene anche alla più piccola subroutine. **Quindi un programmatore deve spesso sprecare parecchio tempo destreggiandosi fra le lunghezze di pagine e le posizioni di memoria dei suoi moduli di programma e delle sue subroutine.**

Prendiamo in considerazione, ad esempio, un programma con i seguenti moduli:

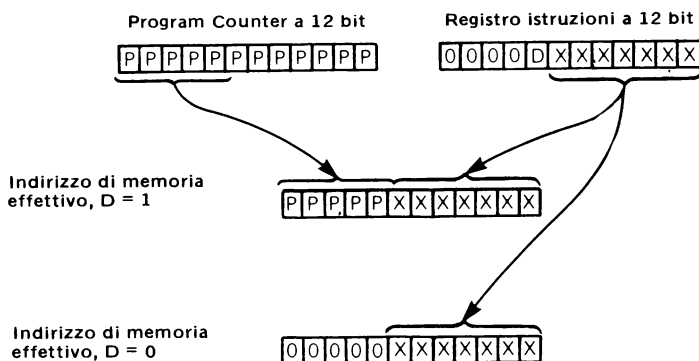
Programma	**Dimensioni (parole)**
MAIN	88_{16}
SUB1	22_{16}
SUB2	78_{16}
SUB3	52_{16}
SUB4	38_{16}
SUB5	50_{16}
SUB6	66_{16}

Possiamo suddividere il programma in questo modo:

Programma	**Dimensioni (parole)**
MAIN	0300_{16} - 0387_{16}
SUB2	0388_{16} - $03FF_{16}$
SUB3	0400_{16} - 0451_{16}
SUB4	0452_{16} - 0489_{16}
SUB5	0490_{16} - $04FF_{16}$
SUB6	0500_{16} - 0565_{16}
SUB1	0566_{16} - 0585_{16}

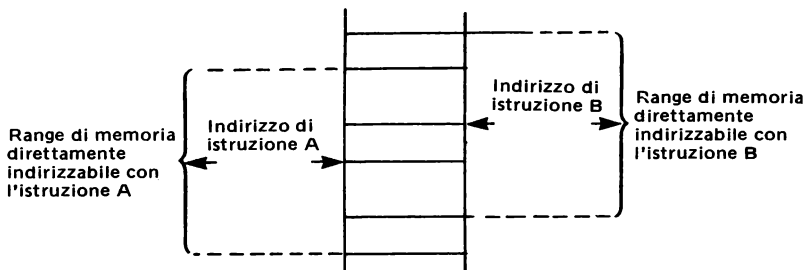
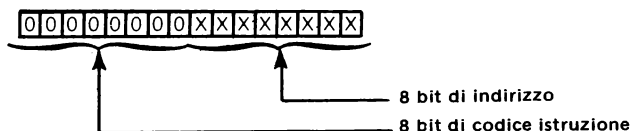
Ma fate attenzione a non commettere errori nella subroutine SUB3, perchè se dovete aumentarne la lunghezza (diciamo di due istruzioni) le subroutine SUB3, SUB4 e SUB5 non staranno più su di una pagina, e la correzione di SUB3 richiederà la risuddivisione dell'intero programma.

PAGINA BASE

Un metodo per eliminare alcune delle restrizioni imposte dallo indirizzamento strutturato a pagine è di dare al computer una pagina base. Questo è quello che fa il PDP-8, quindi esaminiamo il caso specifico. Al fine di offrire a se stesso una scelta in più, il PDP-8 usa sette degli otto bit d'indirizzo per elaborare gli indirizzi all'interno di una pagina: in altre parole, la pagina del PDP-8 non è lunga 256 parole ma solo 128. Comunque, l'ottavo bit vi permette di indirizzare o la pagina in corso, cioè la pagina su cui è posizionata l'istruzione, o la pagina base, cioè una delle 128 parole di memoria. Ve lo illustriamo in questo modo:

Nell'illustrazione precedente, i simboli vengono così usati:

- P rappresenta le singole cifre binarie del Program Counter.
- 0 rappresenta i bit del codice istruzione del registro istruzioni.
- X rappresenta i singoli bit d'indirizzo della parola dell'istruzione.
- D rappresenta il bit di selezione della pagina. Se questo bit è 0, l'indirizzo di memoria effettivo è elaborato spostando i bit X nei bit di ordine inferiore di un registro indirizzi ed inserendo i bit 0 nei cinque bit di ordine superiore del registro indirizzi; in altre parole, si possono indirizzare le posizioni di memoria da 0 a 127. Si dice quindi che si fa riferimento alla pagina base della memoria. Se il bit D è 1, i cinque bit di ordine superiore del registro indirizzi vengono presi dai cinque bit di ordine inferiore del Program Counter; si può far riferimento solo alle posizioni di memoria all'interno della pagina di 128 parole in cui risiede l'istruzione.

IMPAGINAZIONE RELATIVA DEL PROGRAMMA

Una variazione più flessibile dell'impaginazione è l'impaginazione relativa del programma, in cui si suppone che i bit d'indirizzo di un'istruzione rappresentino uno spiazzamento binario con segno, che deve essere aggiunto al contenuto del Program Counter. La pagina relativa del programma può essere illustrata nel modo seguente.

L'indirizzo relativo di programma permette ad un'istruzione di indirizzare la memoria in avanti (positiva) o all'indietro (negativa) con un range in entrambe le direzioni di mezza pagina. Consideriamo ancora un indirizzo di 8 bit, questa volta in una parola di 16 bit:

Supponiamo che il bit d'indirizzo di ordine superiore sia un bit di segno, se l'istruzione è posizionata alla parola di memoria $24AE_{16}$, e gli otto bit d'indirizzo contengono $7A_{16}$, l'indirizzo di memoria effettivo è dato da:

Questo esempio d'indirizzamento di memoria in avanti si può illustrare usando le istruzioni in linguaggio assemblatore, così:

Indirizzo di memoria	Codice oggetto	Istruzione del programma sorgente		
24AD	BC7A		JMP	HERE
—	—		—	
—	—		—	
—	—		—	
2528		HERE	LMA	THERE

Nell'illustrazione precedente, BC sostituisce il codice dell'istruzione JMP. (Il codice oggetto dell'istruzione LMA è irrilevante per la nostra discussione, per cui lo abbiamo lasciato fuori). L'Assembler elaborerà i bit d'indirizzo dell'istruzione JMP sottraendo nel Program Counter dal valore associato alla label HERE, dopo il caricamento della istruzione JMP:

La label HERE è equivalente a	2528 -
Contenuto del PC dopo il caricamento dell'istruzione	24AE =
Differenza	007A

Se l'Assembler elabora un valore maggiore di $7F_{16}$, l'istruzione JMP non è corretta, e l'Assembler stesso trasmetterà un messaggio di errore al programmatore.

Supponiamo che le due istruzioni siano invertire; avremmo:

Indirizzo di memoria	**Codice oggetto**	**Istruzione del programma sorgente**		
24AD		HERE	LMA	THERE
—			—	
—			—	
—			—	
2528	BC84		JMP	HERE

L'Assembler elaborerà i bit d'indirizzo dell'istruzione JMP nello stesso modo, sottraendo il valore al Program Counter, dopo che l'istruzione JMP è stata caricata, dal valore associata alla label HERE:

La label HERE è equivalente a	24AD -
Contenuto del PC dopo il caricamento dell'istruzione	2529 =
Differenza	-7C

— 7C è memorizzato nella forma di complemento a due:

7C	= 0 1 1 1 1 1 0 0	
complemento a uno	= 1 0 0 0 0 0 1 1	
complemento a due	= 1 0 0 0 0 1 0 0	= 84

L'indirizzo di memoria effettivo fornito dall'istruzione JMP viene elaborato nel modo seguente:

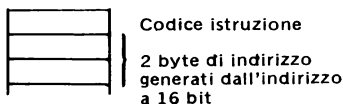
0 0 1 0 0 1 0 1 0 0 1 0 1 0 0 1	2529
1 1 1 1 1 1 1 1 1 0 0 0 0 1 0 0	FF84
0 0 1 0 0 1 0 0 1 0 1 0 1 1 0 1	24AD

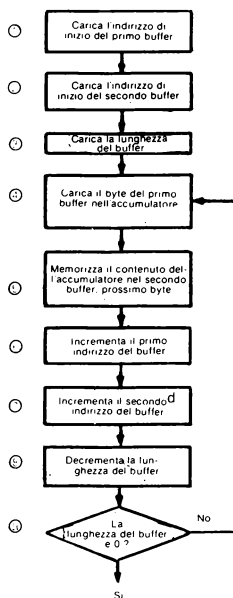
Il bit di segno viene propagato tramite otto bit più significativi

L'INDIRIZZAMENTO DI MEMORIA DIRETTO NEI MICROCOMPUTER

Le variazioni dell'indirizzamento diretto, che sono utili nelle applicazioni dei minicomputer, non sono utili, e spesso non sono nemmeno vitali, nelle applicazioni dei microcomputer. Esaminiamo attentamente il perchè.

INDIRIZZAMENTO DIRETTO ESTESO

Consideriamo prima un'istruzione di tre byte di riferimento diretto alla memoria, che abbiamo così rappresentato:

Il formato dell'istruzione andrà senz'altro bene. I due byte d'indirizzo permettono a qualunque posizione di memoria di essere indirizzata direttamente per memorizzare, leggere o manipolare in altro modo i dati, o per cambiare la sequenza dell'esecuzione del programma con un'istruzione di salto. **Vi sono, comunque, due problemi concernenti l'uso delle istruzioni a tre byte che fanno riferimento diretto alla memoria; primo, i due byte d'indirizzo non possono essere cambiati; secondo, le istruzioni a tre byte consumano molta memoria.**

Ecco un esempio di una sequenza di istruzioni che spostano dei dati da un buffer ad un altro:

Consideriamo il modo con cui la sequenza logica del programma suddetto verrebbe eseguita da un microcomputer che abbia due Data Counter (DC0 e DC1) e due accumulatori (A0 e A1). Supponiamo che il buffer 1 cominci alla posizione di memoria 0470_{16}, il buffer 2 alla posizione di memoria $08C0_{16}$, e che ogni buffer sia lungo otto byte. Illustrando il contenuto del registro dati della CPU, ecco che cosa succede:

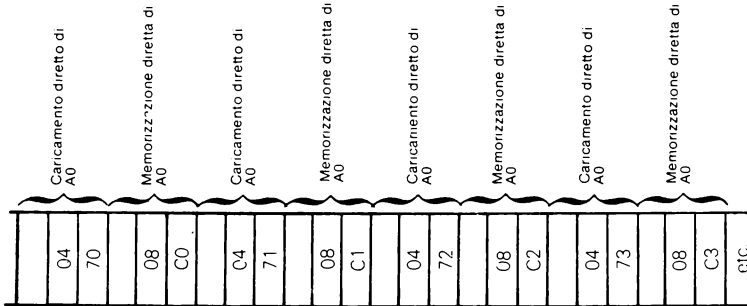
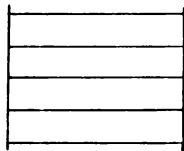
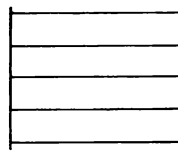
②
04
70
08
C0
08
F9
Carica immediatamente in DC0
Carica immediatamente in DC1
Carica immediatamente in A1
Carica A tramite DC0
Memorizza A tramite DC1
Incrementa DC0
Incrementa DC1
Decrementa A1
Salta se falso
0470
24
7E
62
4A
08
08C1
08C3
08C6
①
04
70
08
C0
08
F9
Carica immediatamente in DC0
Carica immediatamente in DC1
Carica immediatamente in A1
Carica A tramite DC0
Memorizza A tramite DC1
Incrementa DC0
Incrementa DC1
Decrementa A1
Salta se falso
PC
A0
A1
DC0
0470
24
7E
62
1C
4A
01
08
0470
0471
0472
0473
0474
0475
0476
0477
08C0
08C1
08C2
08C3
08C4
08C5
08C6
08C7

4
04
70
08
C0
08
F9
1
Carica immediatamente in DC0
2
Carica immediatamente in DC1
3
Carica immediatamente in A1
4
Carica A tramite DC0
5
Memorizza A tramite DC1
6
Incrementa DC0
7
Incrementa DC1
8
Decrementa A1
9
Salta se falso
PC
A0
A1
DC0
DC1
24
08
0470
08C0
24
7E
62
1C
4A
01
08
41
0470
0471
0472
0473
0474
0475
0476
0477
08C0
08C1
08C2
08C3
08C4
08C5
08C6
08C7
3
04
70
08
C0
08
F9
1
Carica immediatamente in DC0
2
Carica immediatamente in DC1
3
Carica immediatamente in A1
4
Carica A tramite DC0
5
Memorizza A tramite DC1
6
Incrementa DC0
7
Incrementa DC1
8
Decrementa A1
9
Salta se falso
PC
A0
A1
DC0
DC1
08
0470
08C0
24
7E
62
1C
4A
01
08
41
0470
0471
0472
0473
0474
0475
0476
0477
08C0
08C1
08C2
08C3
08C4
08C5
08C6
08C7

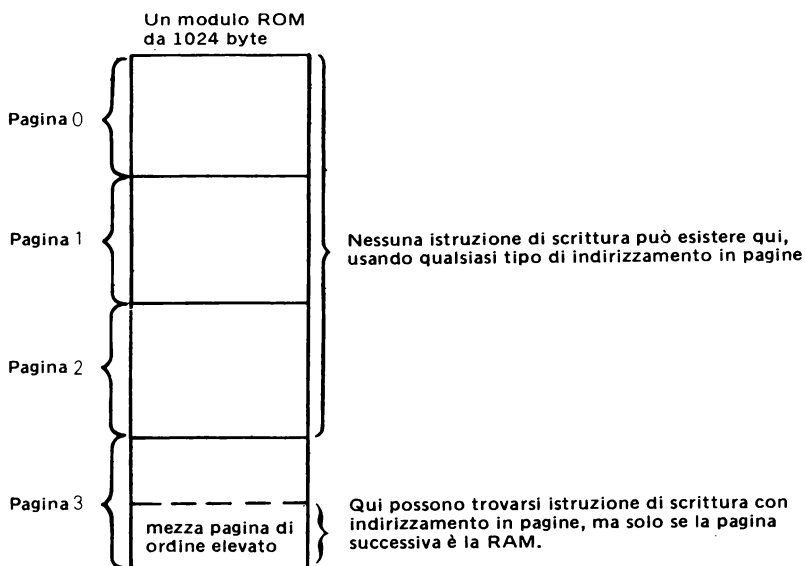
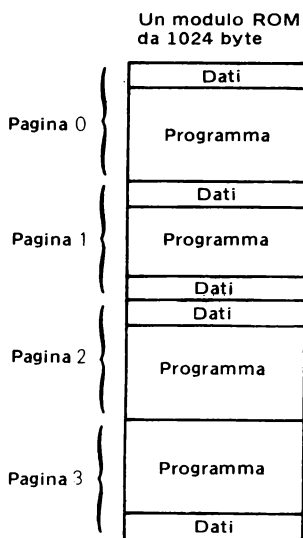
⑤
PC
A0 24
A1 08
DC0 0470
DC1 08C0
04
70
08
C0
08
F1
① Carica immediatamente in DC0
② Carica immediatamente in DC1
③ Carica immediatamente in A1
④ Carica A tramite DC0
⑤ Memorizza A tramite DC1
⑥ Incrementa DC0
⑦ Incrementa DC1
⑧ Decrementa A1
⑨ Salta se falso
24 0470
7E 0471
62 0472
1C 0473
4A 0474
01 0475
08 0476
41 0477
24 08C0
08C1
08C2
08C3
08C4
08C5
08C6
08C7
⑥
PC
A0 24
A1 08
DC0 0471
DC1 08C0
04
70
08
C0
08
F1
① Carica immediatamente in DC0
② Carica immediatamente in DC1
③ Carica immediatamente in A1
④ Carica A tramite DC0
⑤ Memorizza A tramite DC1
⑥ Incrementa DC0
⑦ Incrementa DC1
⑧ Decrementa A1
⑨ Salta se falso
24 0470
7E 0471
62 0472
1C 0473
4A 0474
01 0475
08 0476
41 0477
08C0
08C1
08C2
08C3
08C4
08C5
08C6
08C7

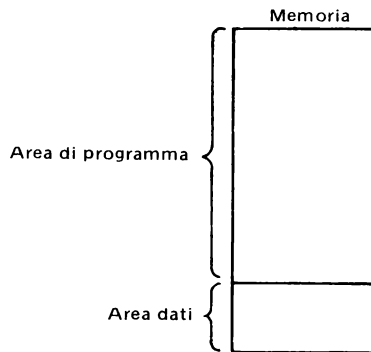
9
04
70
08
C0
08
F9
1 Carica immediatamente in DC0
2 Carica immediatamente in DC1
3 Carica immediatamente in A1
4 Carica A tramite DC0
5 Memorizza A tramite DC1
6 Incrementa DC0
7 Incrementa DC1
8 Decrementa A1
9 Salta se falso
PC
A0 7E
A1 07
DC0 0471
DC1 08C1
24 0470
7E 0471
62 0472
1C 0473
4A 0474
01 0475
08 0476
41 0477
24 08C0
08C1
08C2
08C3
08C4
08C5
08C6
08C7
10
04
70
08
C0
08
F9
1 Carica immediatamente in DC0
2 Carica immediatamente in DC1
3 Carica immediatamente in A1
4 Carica A tramite DC0
5 Memorizza A tramite DC1
6 Incrementa DC0
7 Incrementa DC1
8 Decrementa A1
9 Salta se falso
PC
A0 7E
A1 07
DC0 0471
DC1 08C1
24 0470
7E 0471
62 0472
1C 0473
4A 0474
01 0475
08 0476
41 0477
24 08C0
7E 08C1
08C2
08C3
08C4
08C5
08C6
08C7

Osservate che il programma, come illustrato, sta in 15 byte della memoria di programma. Le istruzioni dalla 4 alla 9 vengono rieseguite otto volte. Questo è possibile, dato che con ogni riesecuzione, la parte che cambia, cioè gli indirizzi della memoria dati, cambia nei Data Counter.

Questo è il vantaggio dell'indirizzamento in memoria implicito. Come verrebbe implementata la stessa logica di programma da un minicomputer che ha indirizzamento diretto, ma non indirizzamento implicito? Se il programma è in ROM, dovrà consistere di una coppia di istruzioni a tre byte ripetute otto volte.

La lunghezza totale del programma sarà di 48 byte. L'indirizzamento in memoria implicito ha fatto risparmiare la memoria permettendo che un set di istruzioni venisse rieseguito, e permettendo alle istruzioni di riferimento alla memoria di occupare un solo byte:

INDIRIZZAMENTO DIRETTO STRUTTURATO A PAGINE

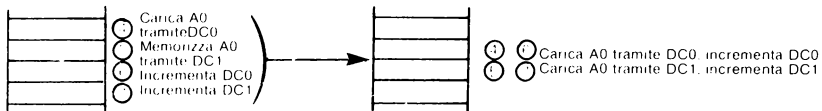
I problemi d'indirizzamento connessi con i microcomputer diventano più rigidi se tentate di usare un qualunque tipo di indirizzamento diretto strutturato a pagine. I progettisti di minicomputer usano l'indirizzamento diretto strutturato a pagine allo scopo di ridurre il numero dei bit d'indirizzo che sono parte di ogni istruzione di riferimento alla memoria. Le pagine possono essere assolute o relative al programma, come abbiamo appena descritto.

Il problema che incontriamo con ogni forma di indirizzamento diretto strutturato a pagine è che esso può essere usato solo per le istruzioni di salto. Questa forma di indirizzamento non può essere usata per le istruzioni che fanno riferimento alla memoria per scrivere. La ROM entra solitamente in moduli di 1024 byte (o più grossi). Le pagine sono lunghe o 128 o 256 byte. Perciò, un'intera pagina sarà o ROM o RAM. Non è possibile avere l'area di programma di una pagina in ROM e l'area dati della stessa pagina in RAM.

Sarebbe certamente possibile avere istruzioni di riferimento alla memoria che leggono i dati dalla memoria di programma, ma ciò richiederebbe la suddivisione della memoria in aree programmi e aree dati, in questo modo:

Questo tipo di complessa suddivisione di memoria aumenta notevolmente i costi di creazione dei programmi per microcomputer e la probabilità di commettere errori

di programmazione. Come risultato, i programmi dei microcomputer sono quasi sempre scritti con aree programmi e dati separate, così:

Questa divisione di memoria in aree di programma e aree dati è talmente generalizzata che alcuni microcomputer specificano memorie separate per i programmi e per i dati. Ciò significa che il Program Counter indirizza una memoria che il Data Counter non può indirizzare e viceversa, il Data Counter indirizza una memoria che il Program Counter non può indirizzare.

AUTOINCREMENTO E AUTODECREMENTO

Nello spostamento di dati che abbiamo appena descritto, osserviamo che due indirizzi, memorizzati nei due Data Counter, devono essere incrementati dopo ogni riferimento alla memoria.

E' facile immaginare la logica di programma che inizia dall'altra estremità di un buffer dati, perciò l'indirizzo di memoria dovrà essere decrementato dopo ogni accesso in memoria. Sia in un caso che nell'altro, **possiamo creare un'istruzione ad un solo byte che specifica un'operazione di riferimento alla memoria, più un incremento o un decremento dell'indirizzo di memoria:**

LO STACK

Esiste una variazione dell'indirizzamento in memoria implicito usata in molti minicomputer ed è implementata in una forma o nell'altra in quasi tutti i microcomputer; è conosciuta come indirizzamento tramite stack. Il concetto di stack è stato presentato alla fine del Capitolo 4, insieme alla logica di indirizzamento dell'unità di controllo del chip slice.

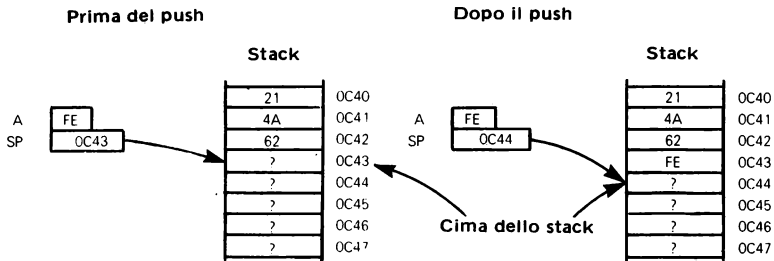
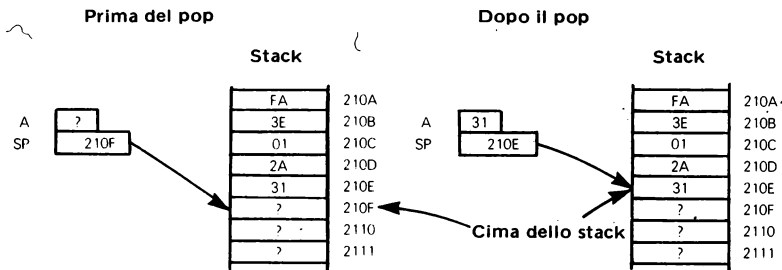
STACK DI MEMORIA

STACK POINTER

L'architettura più comune dello stack usa delle aree di memoria dati per la memorizzazione temporanea di dati ed indirizzi. Lo stack è indirizzato da un registro tipo Data Counter, chiamato stack pointer.

Solitamente sono permesse solo due operazioni sullo stack: lo scrivere sulla parte alta dello stack (chiamata Push, cioè spingere), e la lettura dalla parte alta dello stack (chiamata Pop o Pull, cioè estrarre).

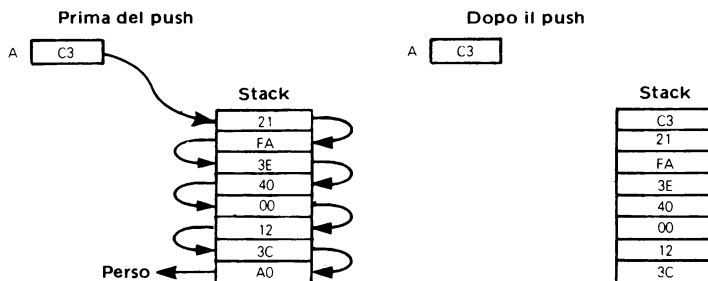
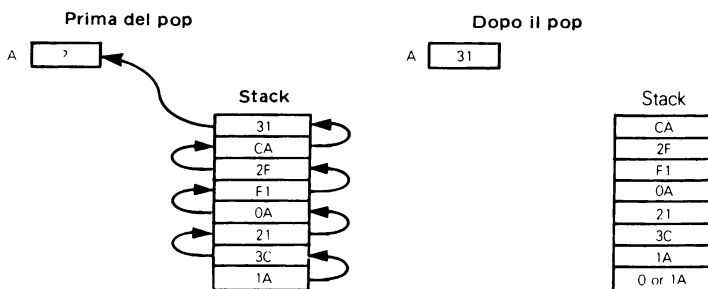
Lo stack prende nome dal fatto che può essere visualizzato come una pila (o stack) di parole dati, in cui si può accedere solo all'ultima parola dati messa nello stack o alla prima parola dati vuota della parte alta dello stack stesso. In entrambi i casi si accede allo stack tramite un indirizzo memorizzato nello stack pointer.

PUSH Un'operazione di push, che scrive nello stack, farà sì che i dati dell'accumulatore (o di qualche altro registro della CPU) vengano scritti nella parola di memoria indirizzata in quel momento dallo stack pointer (SP); il contenuto dello stack pointer viene poi incrementato automaticamente per indirizzare la parola libera successiva, nella nuova parte alta dello stack, in questo modo:

POP Un'operazione di Pull o di Pop è esattamente il contrario di un'operazione di Push; il contenuto del registro dello stack viene decrementato per indirizzare l'ultima parola scritta nella parte alta dello stack, poi il contenuto della parola di memoria indirizzata dallo stack viene spostato nell'accumulatore o in qualche altro registro della CPU. Ecco come possiamo illustrarlo:

Osservate che alla fine di un'operazione di Pop, lo stack pointer si trova ad indirizzare di nuovo la prima parola di memoria libera nella parte alta dello stack; una volta che i dati sono stati letti dalla parte alta dello stack, si suppone che la parola dati sia vuota.

Il parallelo fra l'indirizzamento di memoria implicito, usando un Data Counter, e l'indirizzamento in memoria dello stack, usando uno stack pointer, si dimostra da solo. Infatti la sola differenza fra i due è che il contenuto dello stack pointer DEVE essere incrementato dopo una scrittura, e DEVE essere decrementato dopo una lettura.

Naturalmente, non c'è niente che impedisca ad uno stack di essere implementato nella direzione opposta; cioè, si accede alla parte inferiore dello stack, anzichè a quella superiore, nel senso illustrato prima. Questo stack sarà decrementato dopo una scrittura e incrementato dopo una lettura. Per il resto non cambia niente.

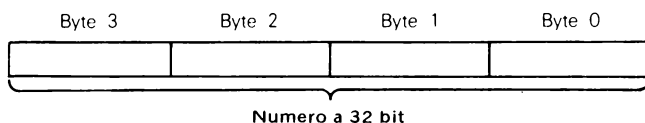
LO STACK IN CASCATA

PUSH Vi è un'architettura dello stack che viene usata in alternativa ma meno frequentemente. Questa architettura fornisce un numero limitato di registri (di solito 8 o 16) nella CPU. Quando un byte dati viene spinto nello stack, avviene quanto segue:

POP Quando un byte dati viene estratto dalla parte alta dello stack, ai dati accade quanto segue:

L'architettura dello stack non richiede stack pointer; dato che in ogni momento i dati vengono scritti nella, o letti dalla cima dello stack.

COME SI USA UNO STACK

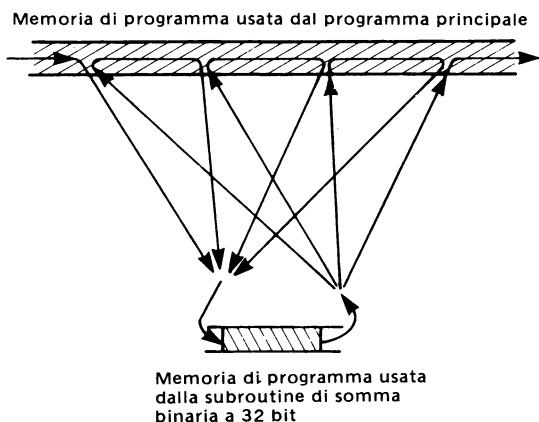
Lo stack è di grande convenienza per gli utenti di minicomputer; è una necessità assoluta nelle applicazioni dei microcomputer.

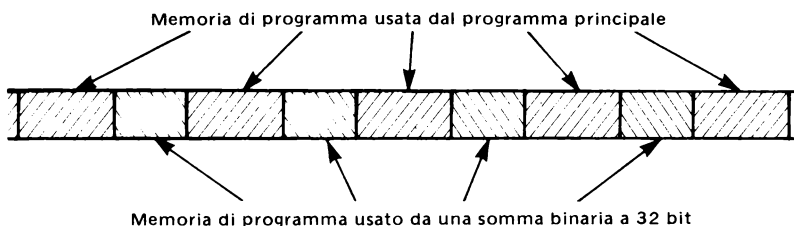
SUBROUTINE **Prendiamo in considerazione l'uso delle subroutine.** La maggior parte dei programmi, sia che siano iscritti per un mini o per un microcomputer, consistono di un numero di sequenze di istruzioni usate di frequente, ognuna delle quali è registrata una volta, in qualche punto della memoria di programma. Si accede allora alla routine come ad una subroutine.

Un'applicazione può richiedere di eseguire calcoli con numeri di 13 bit, che occupano quattro byte contigui, come segue:

Il modo più efficace di trattare questo tipo di calcoli è di scrivere quattro diversi programmi per eseguire addizione, sottrazione, moltiplicazione e divisione a 32 bit. **Ogni volta che volete eseguire l'addizione (per esempio) userete un'istruzione che CHIAMA la subroutine.** Una chiamata si può illustrare in questo modo.

Memoria di programma usata dal programma principale

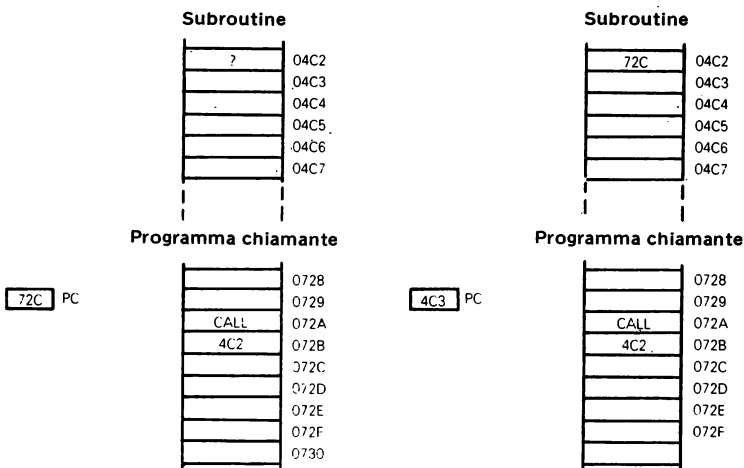
Memoria di programma usata dalla subroutine di somma binaria a 32 bit

→ rappresenta la sequenza di esecuzione dell'istruzione

Supponiamo di non aver usato la subroutine; la sequenza di istruzioni necessarie per eseguire l'addizione binaria a 32 bit avrebbe dovuto essere ripetuta ogni volta che la logica di programma specificava l'addizione binaria a 32 bit. Il programma apparirebbe ora così:

Memoria di programma usato da una somma binaria a 32 bit

La maggior parte dei programmi, sia che siano scritti per i mini che per i microcomputer, possono diventare una grande rete di chiamate delle subroutine. Stabilito che l'importanza delle subroutine in tutti i programmi per microcomputer, viene universalmente accettata, non avete bisogno di capire altro delle subroutine. Comunque, vediamo che cosa succede quando viene chiamata una subroutine, e come la logica di programma tratta il rientro da una subroutine.

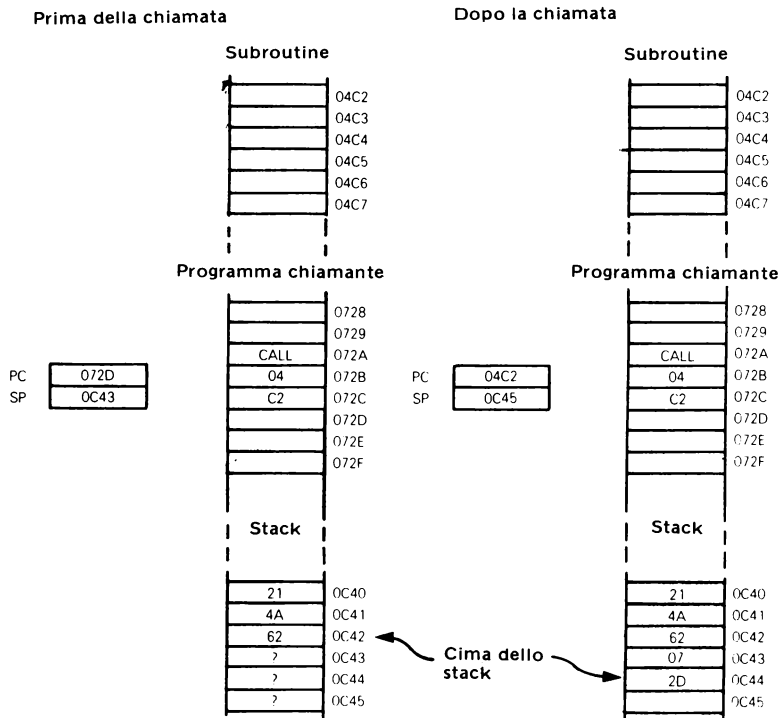
Il minicomputer PDP-8, e alcuni altri vecchi minicomputer usano la prima parola di memoria di una subroutine come posizione in cui deve essere memorizzato l'indirizzo di rientro. Per esempio, supponiamo che le istruzioni della nostra subroutine di addizione binaria a 32 bit occupino le posizioni di memoria da $4C2_{16}$ a $4E0_{16}$. Il PDP-8, essendo un minicomputer a 12 bit, memorizza solo indirizzi di 12 bit. La parola di memoria $4C2_{16}$ è la prima parola della subroutine, e deve essere lasciata vuota. Se la subroutine viene chiamata dalla posizione di memoria $72A_{16}$, si verifica la seguente sequenza di eventi:

1) L'attuale contenuto del Program Counter viene memorizzato nella posizione di memoria $4C2_{16}$.
2) L'indirizzo della prima istruzione della subroutine, $4C3_{16}$, è caricato nel Program Counter.
3) L'esecuzione del programma procede con l'istruzione memorizzata nella posizione di memoria $4C3_{16}$.

Possiamo illustrare tutto ciò in questo modo:

RIENTRO DELLA SUBROUTINE

L'ultima istruzione eseguita all'interno della subroutine deve essere un'istruzione di rientro. Questa istruzione sposta l'indirizzo ($72C_{16}$) memorizzato nella prima parola della subroutine (a $04C2_{16}$) di nuovo nel Program Counter, facendo così in modo che l'esecuzione del programma continui all'istruzione che segue la chiamata della subroutine.

Questo schema per chiamare, le subroutine ovviamente non può andar bene in applicazioni su microcomputer, dato che la subroutine verrà memorizzata nella memoria

di sola lettura; in questo caso, l'indirizzo di rientro non può essere memorizzato nella prima parola della subroutine. I microcomputer memorizzano gli indirizzi di rientro delle subroutine nello stack. La chiamata della subroutine di addizione binaria a 32 bit verrebbe eseguita in questo modo:

Nell'illustrazione precedente, notate, si suppone che le parole di memoria siano di 8 bit. Dato che gli indirizzi sono tutti lunghi 16 bit, sono necessarie due parole di memoria per memorizzare ogni indirizzo.

Per rientrare da una subroutine, è necessario solo estrarre i due byte della parte alta dello stack e metterli nel Program Counter. L'esecuzione procederà poi con l'istruzione che segue la chiamata della subroutine.

SUBROUTINE ANNIDATE E USO DELLO STACK

Una subroutine annidata è una subroutine che è stata chiamata da un'altra subroutine.

SUBROUTINE RICORSIVE

Non vi è niente di insolito nel fatto che una subroutine ne chiami un'altra. Infatti, spesso le subroutine vengono annidate ad un livello di cinque o più. Vi sono perfino certe routine matematiche in cui il modo più efficace di scrivere il programma richiede che la subroutine chiami se stessa. **Una subroutine che è in grado di chiamare se stessa viene chiamata una subroutine ricorsiva.**

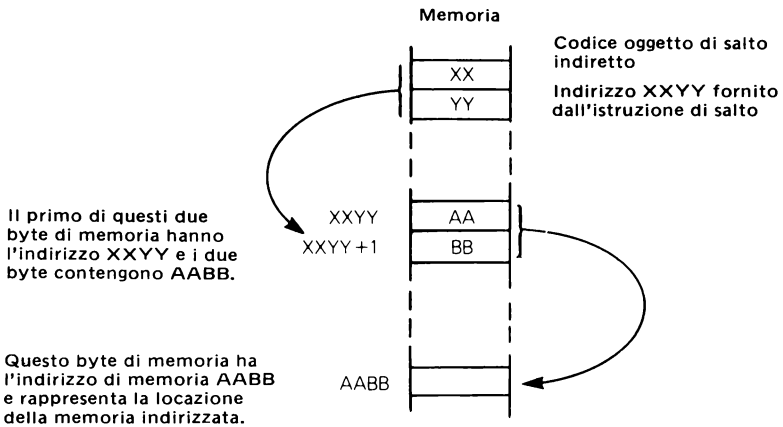
Finchè viene usato lo stack per conservare gli indirizzi di rientro, le subroutine possono essere annidate in qualunque modo, o possono chiamare se stesse; e stabilito che il rientro segua esattamente la chiamata, l'indirizzo di rientro corretto sarà sempre nella parte attualmente alta dello stack.

Se questo libro fosse una guida per il programmatore, dimostreremo ora la suddetta affermazione con estese illustrazioni. Comunque, allo scopo di capire i concetti e la programmazione dei microcomputer, potete prendere per vero il fatto che lo stack assicura che il rientro è esattamente il contrario della chiamata della subroutine.

Se siete ancora curiosi, potete provarlo da soli, definendo un numero di subroutine posizionate in varie aree di memoria. Selezionate arbitrariamente le posizioni all'interno delle subroutine dove vi sono chiamate di altre subroutine. Disegnate uno stack su di un pezzo di carta, e per ogni chiamata mettete nello stack l'indirizzo di rientro. Per ogni rientro, estraete l'indirizzo di rientro e mettetelo nel Program Counter. Vi accorgerete che uscirete dalle subroutine annidate esattamente nella stessa sequenza in cui siete entrati. Per quanto complesse possano essere le sequenze di richiamata della subroutine.

INDIRIZZAMENTO INDIRETTO

INDIRIZZO INDIRETTO

Un indirizzo indiretto specifica una parola di memoria che si suppone contenga l'indirizzo richiesto.

CALCOLO DELL'INDIRIZZO INDIRETTO

Con l'indirizzamento indiretto, l'indirizzo effettivo è dato dall'equazione:

$$EA = (XXYY)$$

dove: EA sta per indirizzo effettivo
( ) rappresentano il contenuto della parola di memoria il cui indirizzo è racchiuso fra le parentesi.

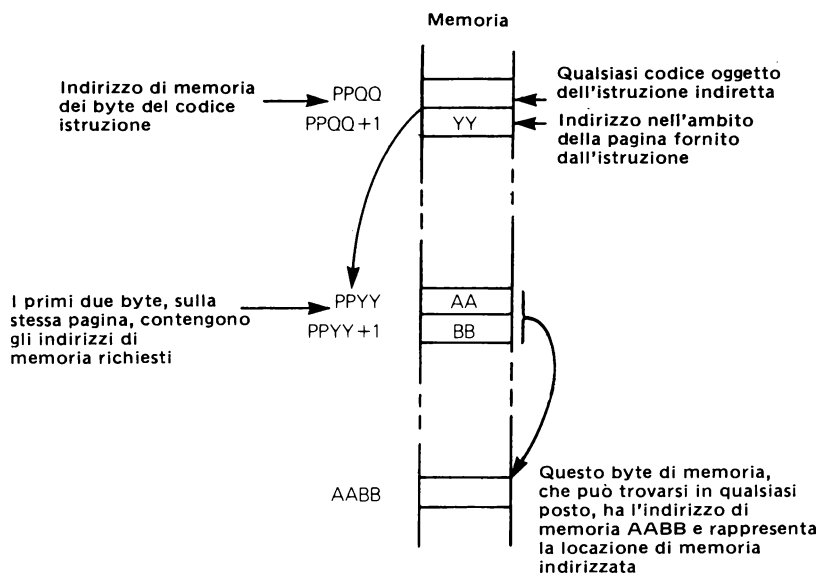
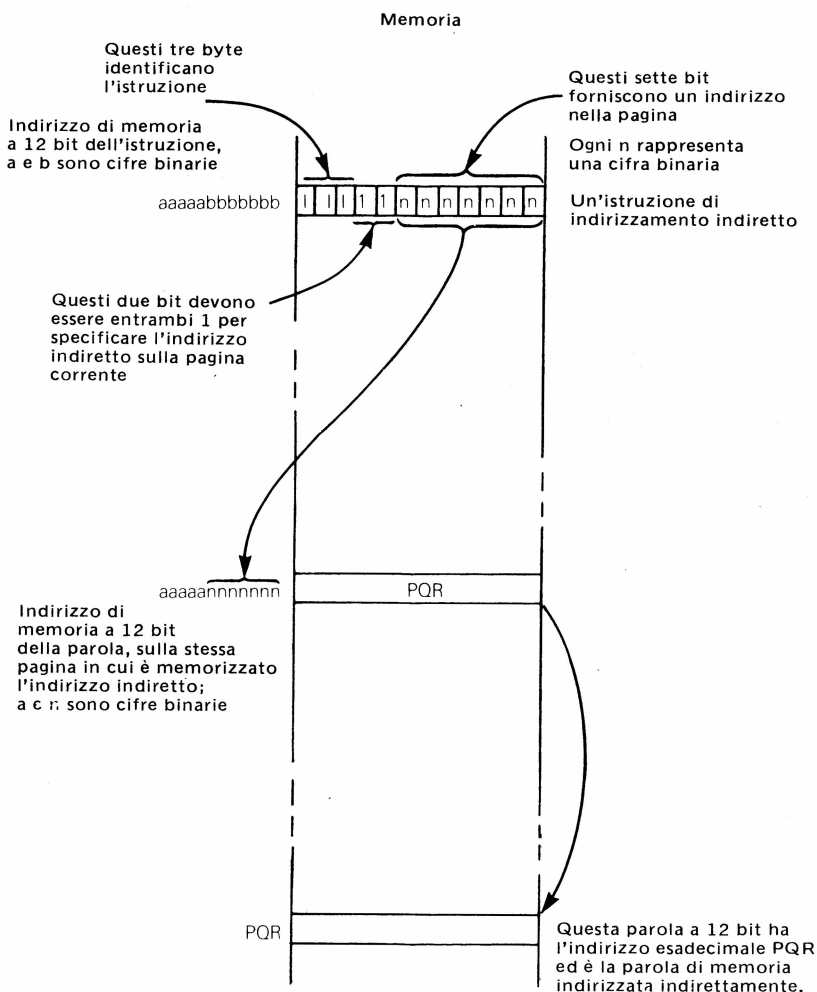
INDIRIZZAMENTO INDIRETTO DI UN COMPUTER CON MEMORIA STRUTTURATA A PAGINE

Indirizzamento indiretto è una necessità assoluta su di un computer strutturato a pagine, dato che il solo modo con cui un programma può accedere ad una posizione di memoria all'esterno della pagina di un'istruzione. Perciò, su di un computer con memoria strutturata a pagine, l'istruzione dell'indirizzamento diretto

LMA XXYY

deve, se XXYY è al di là della pagina dell'istruzione, essere sostituita dall'istruzione dell'indirizzamento indiretto:

nell'istruzione LMA (Caricare nell'accumulatore), HERE fornisce l'indirizzo della parola di memoria dell'indirizzo indiretto e "*" specifica l'indirizzamento indiretto. La posizione di memoria HERE contiene un indirizzo XXYY che diventa l'indirizzo di memoria effettivo.

In un microcomputer, con memoria strutturata a pagine, un'istruzione di indirizzamento indiretto occuperà solo due byte. L'indirizzo di memoria effettivo viene allora calcolato in questo modo:

Il PDP-8, essendo un minicomputer a 12 bit, usa la seguente variazione dell'indirizzamento indiretto strutturato a pagine:

INDIRIZZAMENTO INDIRETTO TRAMITE LA PAGINA BASE

Il PDP-8, e ogni altro computer con una pagina base, usano una gran parte della pagina base per memorizzare gli indirizzi; a questi indirizzi si farà riferimento indirettamente. Per esempio, supponiamo che nell'illustrazione precedente, il codice oggetto alla posizione di memoria aaaaabbbbbbb sia (III 10nnnnnnn), invece di essere III 11nnnnnnn. Ora andrebbe scelto l'indirizzo memorizzato nella posizione di memoria 00000nnnnnnn, non l'indirizzo memorizzato nella posizione di memoria aaaaannnnnnn.

AUTOINCREMENTO E DECREMENTO INDIRETTO

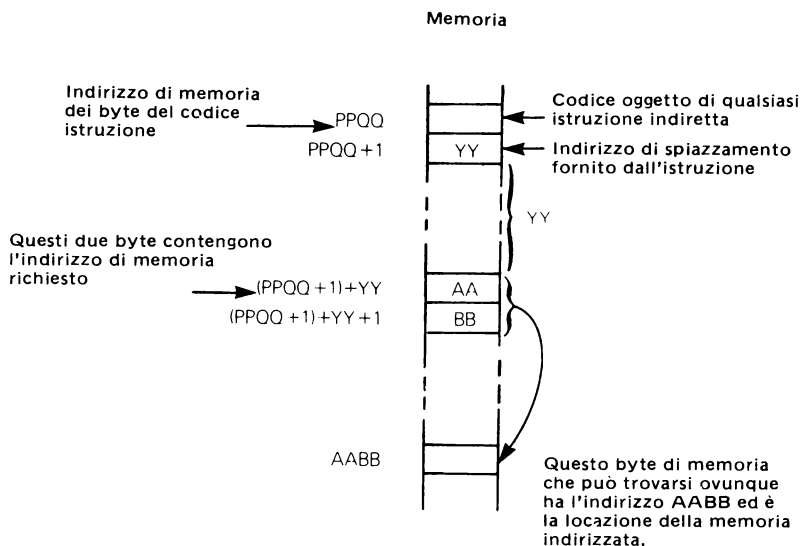
Un'altra variazione dell'indirizzamento usa certe posizioni di memoria come posizioni di autoincremento o decremento. Per esempio, il minicomputer PDP-8 usa le posizioni di memoria da 008_{16} a $00F_{16}$ sulla pagina base, come posizioni di autoincremento. Se viene memorizzato un indirizzo in qualunque posizione di autoincremento, l'indirizzo sarà incrementato in qualunque momento si fa riferimento ad esso in modo indiretto.

Con riferimento all'ultima illustrazione, se aaaaannnnnnn, fosse 000000001000 (cioè 008_{16}), dopo che è stata eseguita l'istruzione di indirizzamento indiretto, la posizione di memoria 008_{16} conterrebbe PQR + 1; all'esecuzione successiva dell'istruzione di indirizzamento indiretto, PQR + 1 sarebbe l'indirizzo di memoria effettivo, e non PQR.

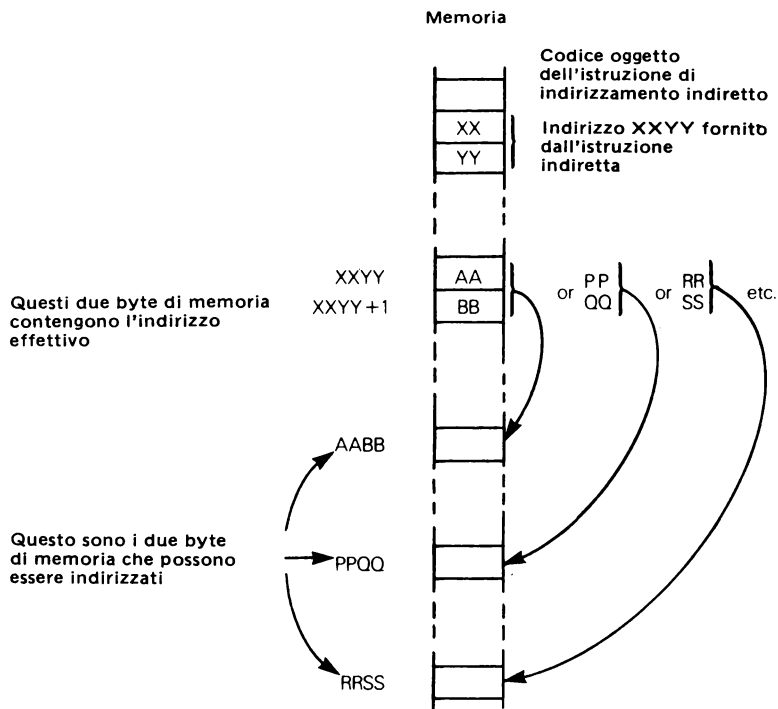
Un indirizzo indiretto ad autodecremento avrebbe generato PQR – 1 invece di PQR + 1.

Osservate che la pagine base del PDP-8 deve essere implementata nella memoria di lettura-scrittura se bisogna cambiare gli indirizzi memorizzati ivi contenuti.

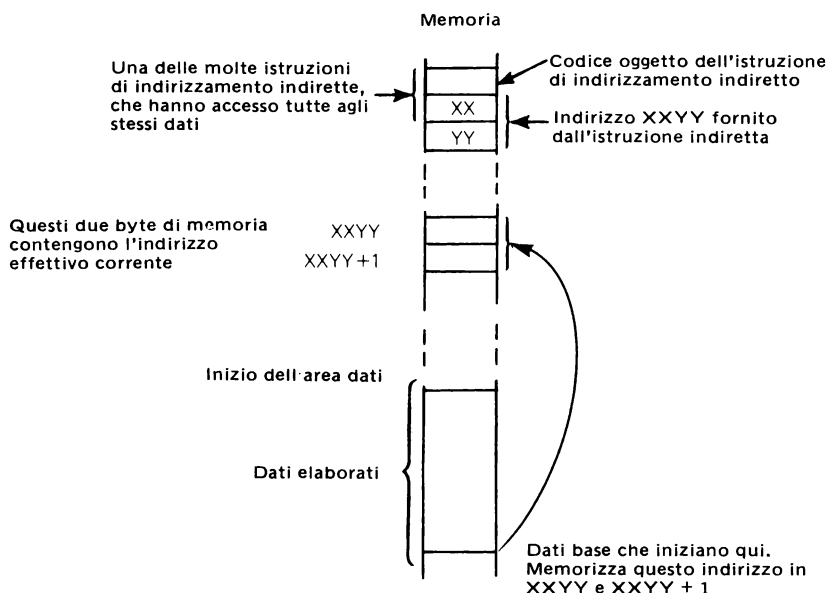
INDIRIZZAMENTO INDIRETTO RELATIVO AL PROGRAMMA

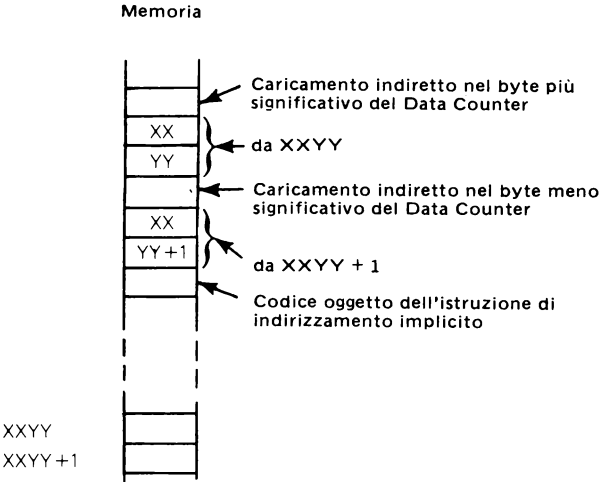
Un computer che usa l'indirizzamento diretto, relativo al programma può avere anche l'indirizzamento indiretto relativo al programma. Ecco che cosa succede:

INDIRIZZAMENTO INDIRETTO – CONFRONTO TRA MINI E MICROCOMPUTER

Nel mondo dei minicomputer, anche se l'indirizzamento diretto non è strutturato a pagine, l'indirizzamento indiretto è di grande convenienza, dato che permette ad un'istruzione di caricamento o di memorizzazione di accedere ad un certo numero

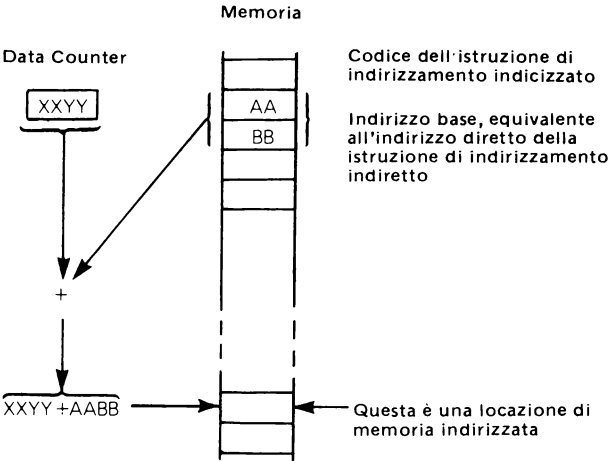
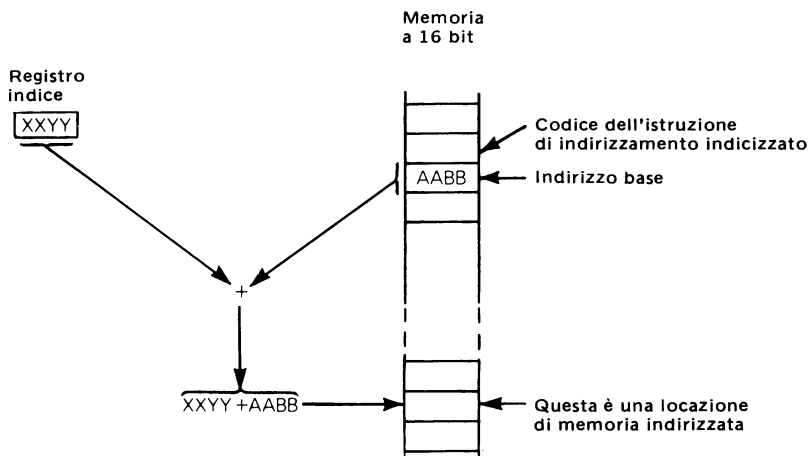
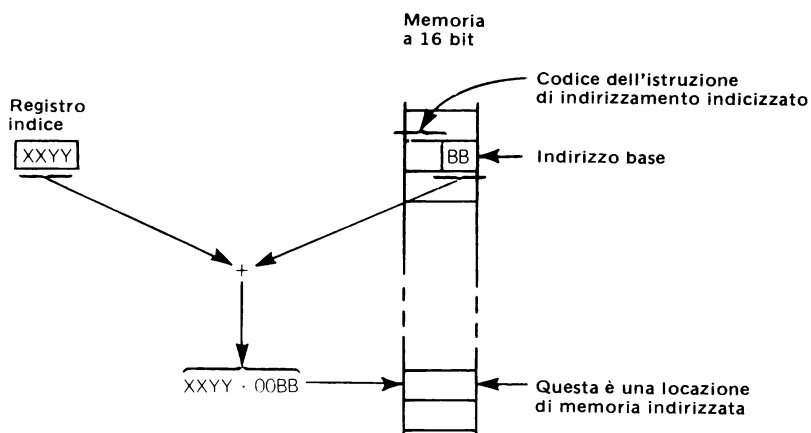
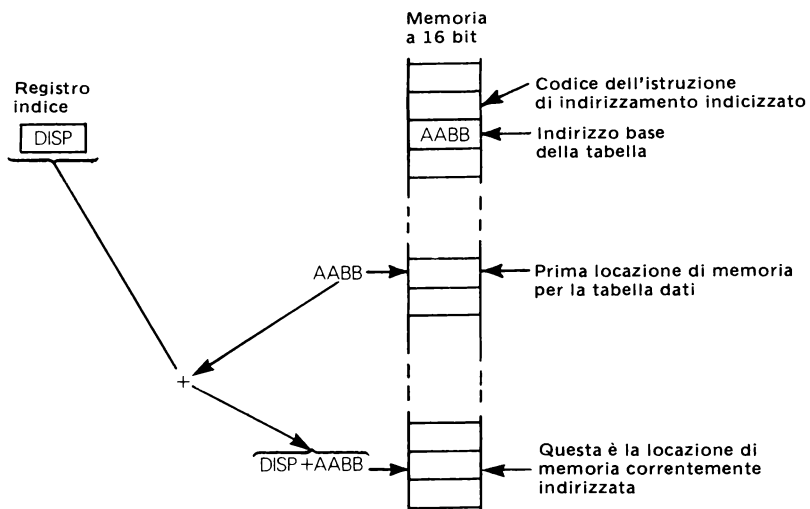
di posizioni di memoria diverse, a seconda del contenuto corrente della parola di indirizzamento di memoria indiretto. Vediamo l'esempio seguente:

Perchè vorreste cambiare l'indirizzo indiretto AABB in PPQQ o RRSS?

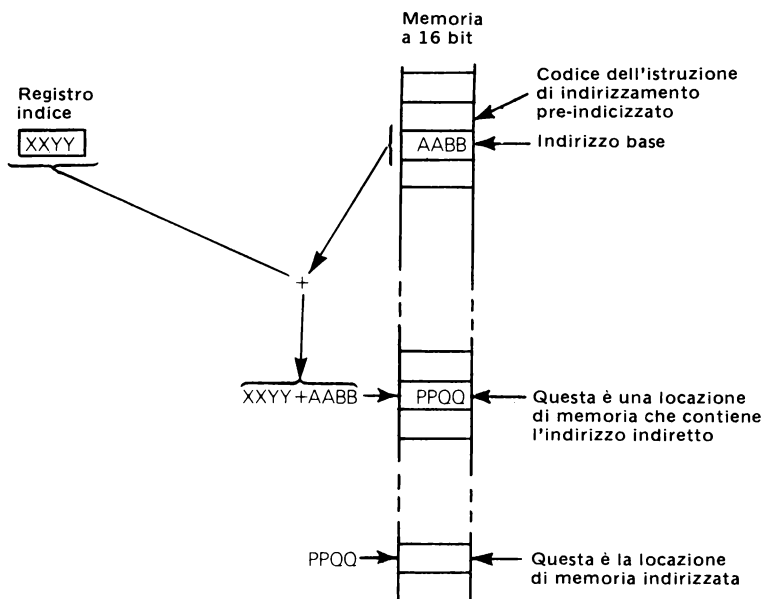
Molti minicomputer operano a divisione di tempo. Ciò significa che un solo minicomputer può eseguire vari programmi, dedicando ad ognuno pochi millisecondi prima di passare al successivo. Ogni programma userà parti di memoria per memorizzare i programmi e i dati che sono necessari per l'esecuzione immediata, mentre il grosso dei programmi e dei dati rimane su disco. Un programma o una tabella dati può occupare completamente diverse aree di memoria in ogni riesecuzione. Questo accade perchè l'area di memoria che è libera per l'uso può essere compressa da programmi che non hanno niente a che vedere e che venivano eseguiti, dal sistema a divisione di tempo, nei millisecondi precedenti. **L'indirizzamento indiretto variabile è uno dei modi con cui i minicomputer sono capaci di gestire il fatto che i programmi e le tabelle dati possono occupare aree diverse di memoria da una esecuzione all'altra.** E' necessario solo cambiare alcuni indirizzi, come AABB, al fine di cambiare la posizione di una tabella dati o di un programma. **Mentre questa giustificazione dell'indirizzamento indiretto è molto significativa nelle complesse applicazioni dei minicomputer, non ha assolutamente senso nelle applicazioni dei microcomputer.** Quando il sistema globale di un microcomputer, completo di memoria, costa un centinaio di dollari o meno, sarà più economico dare ad ogni utente la sua CPU e la sua memoria, invece di affrontare le spese di programmazione richieste per suddividere l'uso di un articolo di così basso costo come è un microcomputer.

Anche le applicazioni senza divisione di tempo possono fare uso in maniera efficace dell'indirizzamento indiretto variabile. Per esempio, una sola area dati può essere usata da più tabelle dati di lunghezza variabile. Consideriamo una semplice applicazione riguardante le telecomunicazioni. I dati sono in arrivo su di una linea telefonica ad una velocità casuale; quando essi arrivano, vengono memorizzati nella memoria di lettura-scrittura. A intervalli fissi, viene eseguito un programma per elaborare il gruppo più recente di segmenti di dati. Osservate che ogni volta che il programma viene eseguito, i dati su cui esso deve operare risiederanno in un'area diversa di memoria di lettura-scrittura. Se il programma fa riferimento alla memoria di lettura-scrittura per mezzo dell'indirizzamento indiretto, allora, caricando semplicemente il nuovo indirizzo iniziale dell'area dati nello spazio dell'indirizzo indiretto, può accedere al seguente segmento di dati — in qualunque posto si trovi:

Naturalmente XXYY e XXYY + 1 deve essere una locazione di memoria a lettura-scrittura.

Un minicomputer che ha solo l'indirizzamento implicito e diretto dovrebbe simulare l'indirizzamento indiretto per eseguire le operazioni suddette. Si potrebbe farlo usando tre istruzioni per ogni istruzione di indirizzamento indiretto, così:

INDIRIZZAMENTO INDICIZZATO

Alcuni costruttori di microcomputer descrivono l'indirizzamento di memoria implicito come indirizzamento indicizzato; essi sono simili ma non uguali.

Un indirizzo indicizzato è la somma di un indirizzo diretto e di uno implicito. Possiamo illustrarlo in questo modo:

REGISTRO INDICE **Quando viene usato nel modo precedentemente illustrato, il Data-Counter viene chiamato registro indice.** Alcuni minicomputer non hanno registro indice, o indirizzamento indicizzato, ma la maggior parte sì. I minicomputer che hanno indirizzamento indicizzato possono avere da 1 a 15 registri indici. L'indirizzo indicizzato dei minicomputer a 16 bit può essere illustrato nel modo seguente, per istruzioni lunghe:

Per le istruzioni brevi, l'indirizzamento indicizzato nei minicomputer a 16 bit si può illustrare così:

Il registro indice prende questo nome dal fatto che è probabile che il suo contenuto cambi (per esempio, mentre esso indicizza un'area di memoria che viene considerata come una tabella dati). Per un minicomputer a 16 bit, possiamo illustrarlo come segue.

PRE-INDICIZZATO

L'indirizzamento indicizzato può essere combinato con l'indirizzamento indiretto, e ciò dà adito a due possibilità; l'indice può essere applicato all'indirizzo base, o all'indirizzo indiretto. **Quando l'indice è applicato all'indirizzo base, parliamo di indirizzamento pre-indicizzato.** Lo illustriamo così:

INDIRIZZO EFFETTIVO

Per l'indirizzamento pre-indicizzato, l'indirizzo effettivo (EA) è dato dall'equazione:

EA = (BASE + INDICE)

Le parentesi significano "contenuto di". Nell'illustrazione precedente:

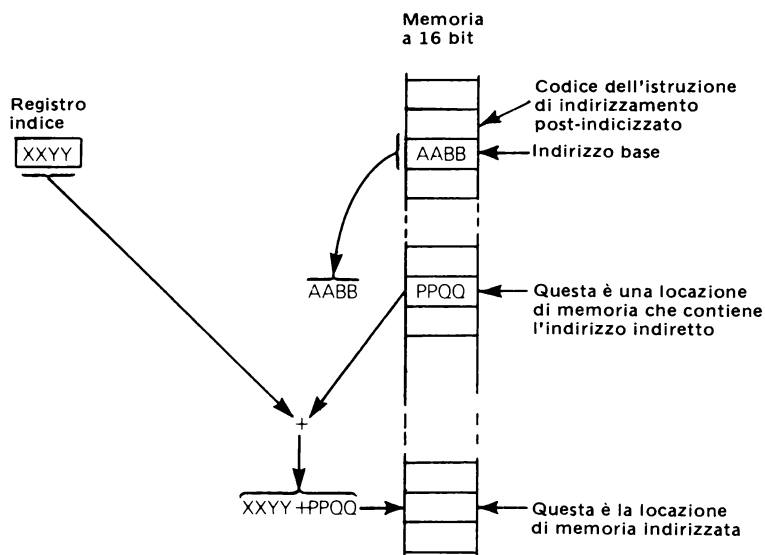
EA = (AABB + XXYY) = PPQQ

Supponiamo che il registro indice contenga $213A_{16}$ e l'indirizzo base sia $413C_{16}$, l'indirizzo effettivo è dato da:

$$EA = [413C_{16} + 213A_{16}] = [6276_{16}]$$

Perciò l'indirizzo effettivo è il contenuto della parola di memoria con indirizzo 6276_{16}.

POST-INDICIZZATO

Quando l'indice viene applicato all'indirizzo indiretto, parliamo di indirizzamento post-indicizzato. Ecco come si può illustrare:

INDIRIZZO EFFETTIVO

Per l'indirizzamento post-indicizzato, l'indirizzo effettivo (EA) è dato dall'equazione:

EA = (BASE) + INDICE

Le parentesi significano di nuovo "contenuto di". Nell'illustrazione precedente:

EA = [AABB] + XXYY = PPQQ + XXYY

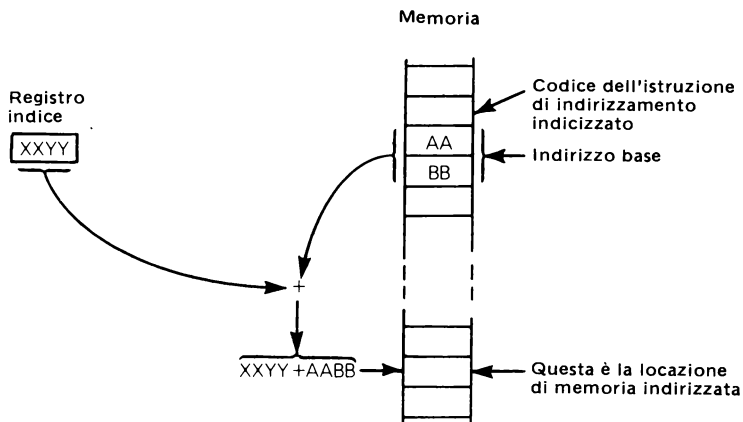
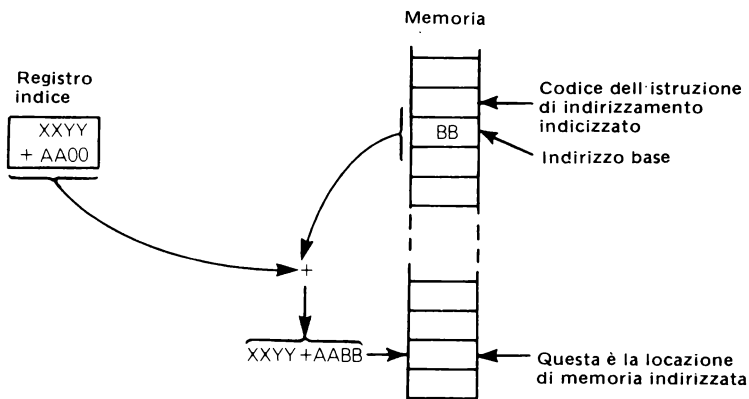
Supponiamo di nuovo che il registro indice contenga $213A_{16}$ e l'indirizzo base sia $413C_{16}$; l'indirizzo effettivo è dato da:

$$EA = [413C_{16}] + 213A_{16}$$

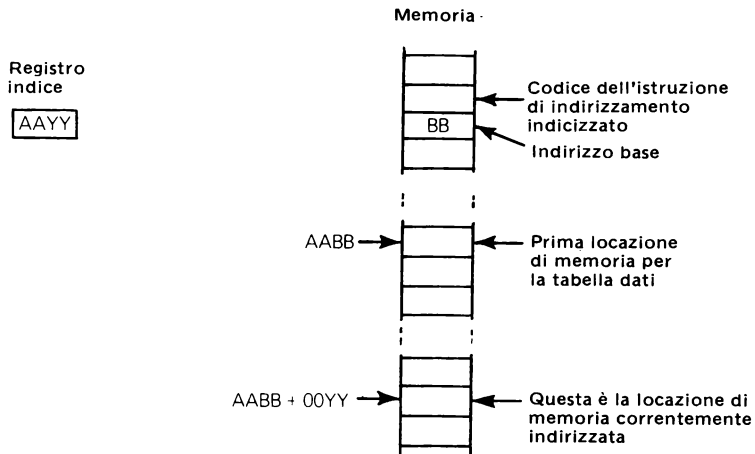
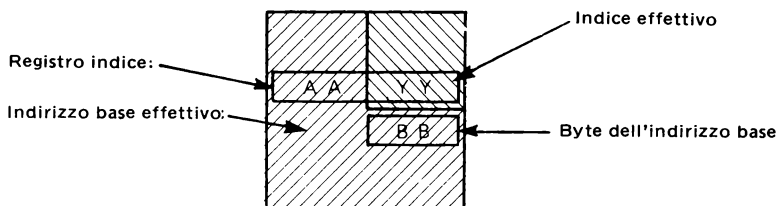
Perciò l'indirizzo effettivo è il contenuto della parola di memoria con indirizzo $413C_{16}$, più $213A_{16}$.

INDIRIZZAMENTO INDICIZZATO NEI MICROCOMPUTER

La lunghezza di parola di 8 bit è la caratteristica dei microcomputer che stabilisce quanto indirizzamento indicizzato, se c'è, verrà implementato.

Incominciamo con l'osservare l'indirizzamento indicizzato nella sua forma più semplice. Per un microcomputer a 8 bit, possiamo illustrarlo così:

Chiaramente vi è un po' di ridondanza nell'indirizzamento indicizzato, come precedentemente illustrato: XXYY + AABB non può essere più di $FFFF_{16}$, dato che questo è il valore massimo che un indirizzo di 16 bit può avere. Perciò ogni indirizzo indicizzato può essere in questo modo:

Questo è l'indirizzo effettivo:

EA = XXYY + AA00 + 00BB
= XXYY + AABB

Abbiamo risparmiato un byte nella nostra istruzione di indirizzamento indicizzato, e non abbiamo dimenticato niente. In termini di tabelle dati indicizzate, questa rappresentazione di indirizzamento si può illustrare così:

Dato che abbiamo un registro indice di 16 bit, ma una parola di memoria di soli 8 bit, quello che stiamo in effetti facendo, è creare l'indirizzo base della tabella derivandolo dal byte di ordine superiore del registro indice, più i byte dell'indirizzo base. Il byte di ordine inferiore del registro indice diventa l'indice della tabella.

Nel mondo dei microcomputer, l'indirizzamento indicizzato diretto si trova raramente. Questo succede perchè il codice istruzione è di 8 bit e, se tentiamo di specificare troppe scelte di indirizzamento, consumeremo velocemente tutti i bit.

Capitolo 7
UN SET DI ISTRUZIONI

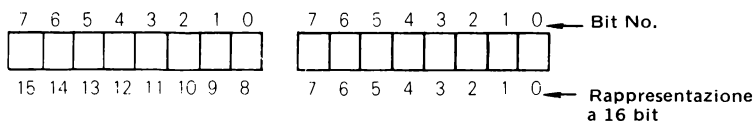
Siamo ora pronti per creare un ipotetico set di istruzioni. Il set di istruzioni che stiamo per creare non copierà nessun set di istruzioni per microcomputer esistente. Piuttosto, conterrà delle caratteristiche che li rappresenteranno tutti; ciò che dobbiamo fare è giustificare ognuna di queste caratteristiche.

ARCHITETTURA DELLA CPU

Il primo prerequisito, prima di poter discutere delle singole istruzioni, è la selezione del numero e del tipo di registri, nonchè del numero e del tipo di modi d'indirizzamento che il nostro ipotetico microcomputer avrà. Incominceremo con i registri.

NUMERO DEI REGISTRI

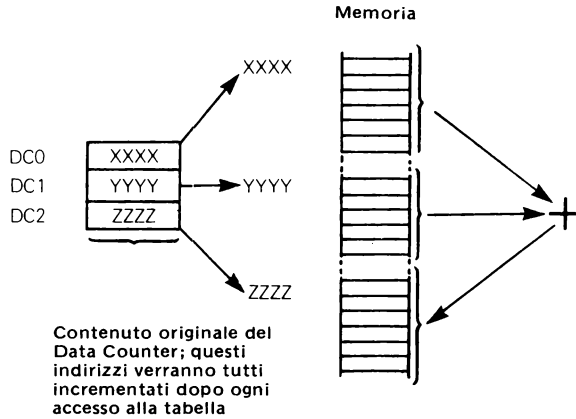
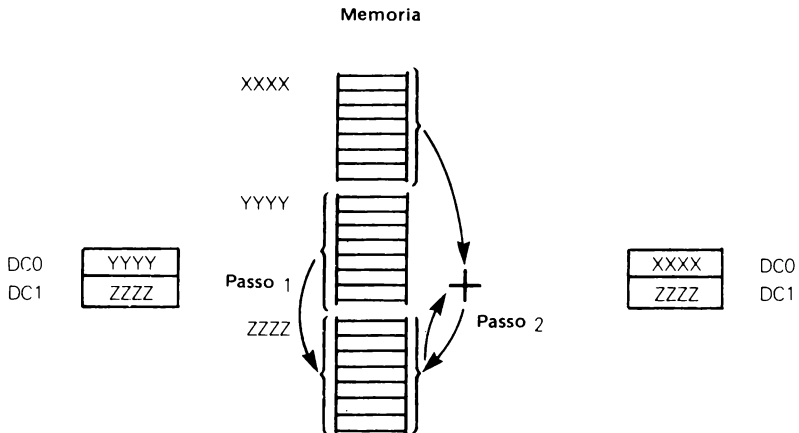
Non possiamo semplicemente decidere di avere un grande numero di registri — più che sufficienti per qualunque situazione. Ricordate, ogni registro deve diventare logica del chip, consumando lo spazio limitato del chip della CPU; inoltre, se abbiamo molti registri, dovremmo usare molti bit del codice istruzione, solo per identificare il registro a cui dobbiamo far riferimento. Perciò, dobbiamo giustificare attentamente ogni singolo registro che decidiamo di avere.

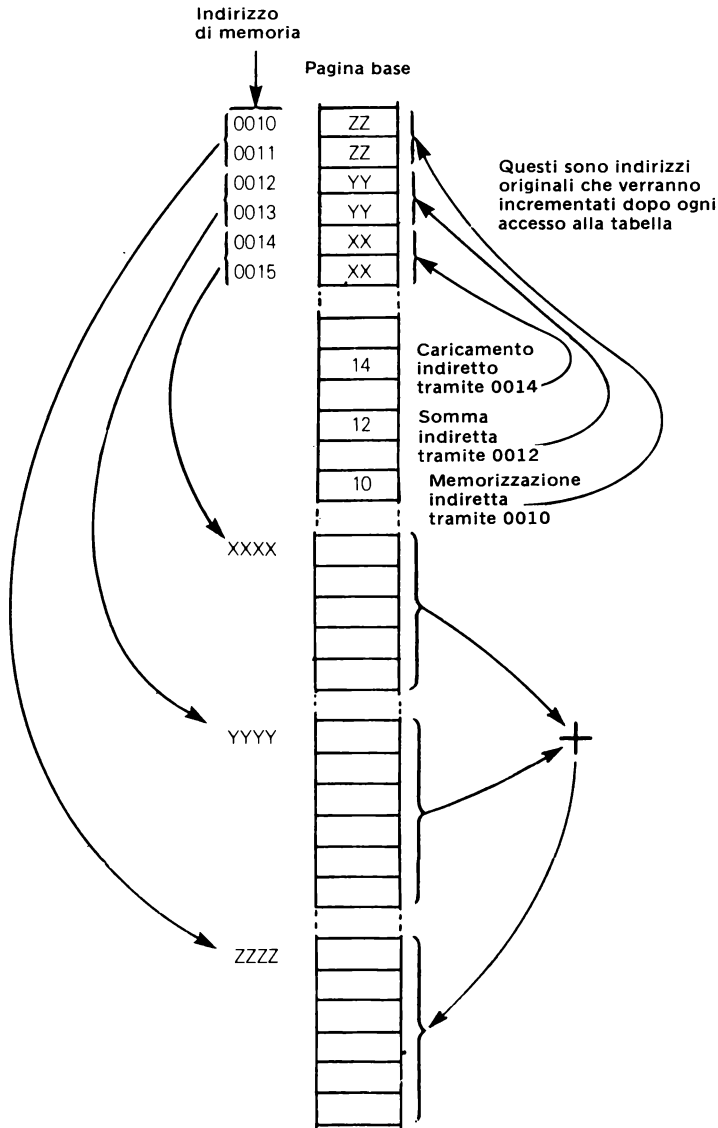
Scegliamo due accumulatori (A0 e A1). Il fatto di avere due accumulatori al posto di uno semplifica le operazioni sui dati di 16 bit, dato che due accumulatori possono essere visualizzati come il byte superiore e il byte inferiore di una sola unità di 16 bit:

Le operazioni con dati di 16 bit si incontrano abbastanza spesso da giustificare il fatto di avere due accumulatori.

L'avere due accumulatori è utile anche quando si devono leggere dei dati da due tabelle ed elaborarli in parallelo; ciò avviene più facilmente e più velocemente con due accumulatori, che, in effetti, forniscono due canali indipendenti per il trasferimento dei dati.

DATA COUNTER

Saranno sufficienti due accumulatori? Alcuni microcomputer hanno più di due accumulatori, o registri equivalenti, ma i registri addizionali vengono usati come Data Counter o registri d'indirizzo di memoria di qualche tipo. **Forniremo al nostro microcomputer tre Data Counter di 16 bit (DC0, DC1 e DC2)**, perciò, non abbiamo bisogno di più di due accumulatori, o di registri equivalenti.

Perchè abbiamo tre Data Counter? La risposta è che ciò semplifica notevolmente l'uscita dei dati di elaborazione da tabelle.
Spesso i dati di due tabelle sorgente vengono combinati in qualche modo (l'esempio più ovvio è l'addizione multi-byte) e il risultato è immagazzinato in una terza tabella.
I microcomputer con meno di tre Data Counter, o registri di indirizzo equivalenti devono trasferire gli indirizzi fra l'immagazzinamento temporaneo e la memoria, o devono diversamente superare le limitazioni di avere solo uno (o due) Data Counter.
Consideriamo il semplice caso di addizione a più byte, avendo tre Data Counter, questa operazione, e quelle similari, si possono facilmente trattare in questo modo:

Avendo due Data Counter, dovremmo spostare uno dei buffer dati sorgente (iniziando o con XXXX o con YYYY) nel buffer del risultato, poi addizionare in questo modo:

Un microcomputer che abbia solo un Data Counter dovrebbe memorizzare i tre indirizzi delle tabelle in qualche punto della memoria di lettura-scrittura, poi caricare ogni indirizzo nel Data Counter prima di accedere ad ogni tabella.

Un microcomputer con indirizzamento indiretto potrebbe memorizzare i tre indirizzi delle tabelle nella memoria di lettura-scrittura, poi accedere indirettamente alle tabelle per mezzo dei tre indirizzi:

SOMMARIO DEI REGISTRI DELLA CPU

Daremo al nostro computer uno Stack Pointer (SP) ed un Program Counter (PC). La nostra batteria di registri appare come illustrato nella figura a pagina seguente.

8-bit	Accumulatore AC0
8-bit	Accumulatore AC1
16-bit	Data Counter DC0
16-bit	Data Counter DC1
16-bit	Data Counter DC2
16-bit	Stack Pointer SP
16-bit	Program Counter PC

FLAG DI STATO

Nel Capitolo 2 abbiamo descritto i flag di stato, e il modo in cui si usano. Forniremo al nostro ipotetico computer i quattro flag di stato Z (Zero), C (Carry), O (Overflow) e S (Sign).

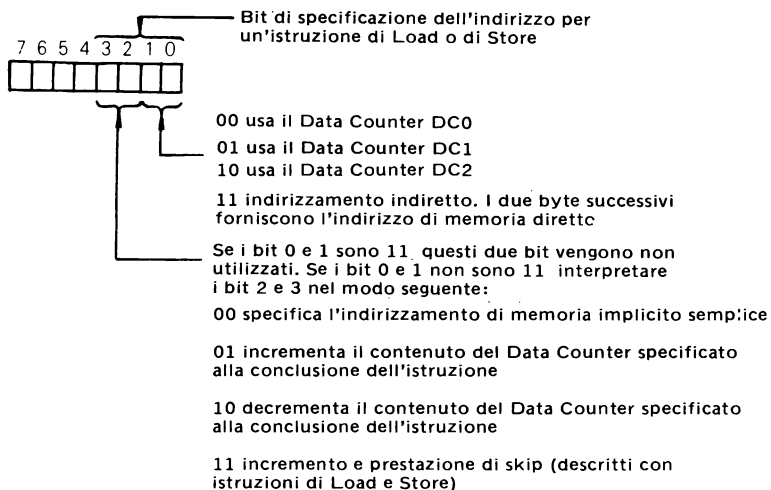
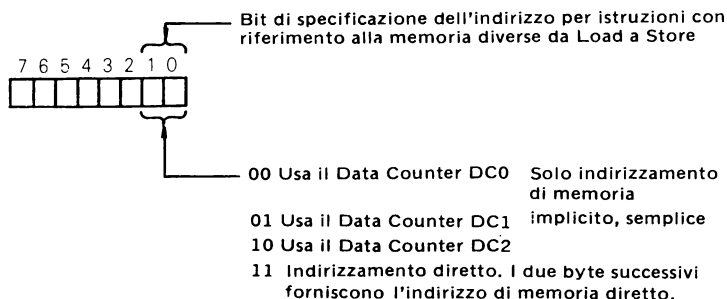
MODI DI INDIRIZZAMENTO

I modi di indirizzamento di un microcomputer dipendono dal numero e dal tipo di registri che abbiamo scelto.

Per esempio, un microcomputer che ha un solo Data Counter probabilmente userà l'indirizzamento indiretto come mezzo alternativo per accedere simultaneamente ad un certo numero di aree dati; di questo abbiamo parlato trattando l'argomento dei Data Counter.

Dato che abbiamo tre Data Counter, risparmieremo sui bit del codice istruzione e sulla logica della CPU, non avendo indirizzamento indiretto; piuttosto, includeremo le possibilità di autoincremento e autodecremento. **Le istruzioni che si riferiscono alla memoria saranno divise in due categorie, come segue:**

1) **Istruzioni di Load (caricamento) e di Store (memorizzazione).** Dato che vengono usate frequentemente, queste due istruzioni avranno un set completo e flessibile di scelte di indirizzamento.
2) **Altre istruzioni di riferimento alla memoria,** essendo meno comunemente usate, avranno capacità di indirizzamento di memoria più limitate.

La capacità di indirizzamento della memoria delle istruzioni di Load e di Store può essere rappresentata in questo modo:

Le istruzioni di riferimento alla memoria diverse da Load e Store, offriranno questo sotto-test limitato di scelte d'indirizzamento:

DESCRIZIONE DELLE ISTRUZIONI

Vi sono due tendenze, opposte fra di loro, che dobbiamo tener presenti nel valutare le istruzioni che vanno a costituire il set di istruzioni del nostro microcomputer. Prima, dobbiamo decidere quali tipi di istruzioni sono essenziali, molto utili, o semplicemente desiderabili; poi, dobbiamo selezionare le istruzioni che usano le 256 possibili combinazioni fornite da un codice istruzione di 8 bit; oltre a queste non possiamo avere altre scelte.

Al fine di equilibrare le due tendenze nella discussione che segue, creeremo un set di istruzioni completo, ma ipotetico. Ciò significa che dobbiamo giustificare ogni istruzione, o tipo di istruzione, e dobbiamo specificare la configurazione del codice oggetto che deve essere interpretata dall'unità di controllo della CPU quando identifica l'istruzione.

ISTRUZIONI DI INPUT/OUTPUT

Un microcomputer non sarebbe utile se non fornisce i mezzi per ricevere dati dai, e trasmettere dati ai dispositivi esterni; questo è l'input/output e viene guidato per mezzo delle istruzioni di input/output (I/O).

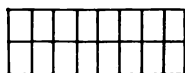
Un'istruzione di I/O ha bisogno di specificare tre cose:

1) **L'istruzione sta leggendo i dati da un dispositivo esterno (input) o sta trasmettendo i dati ad un dispositivo esterno (output)?**

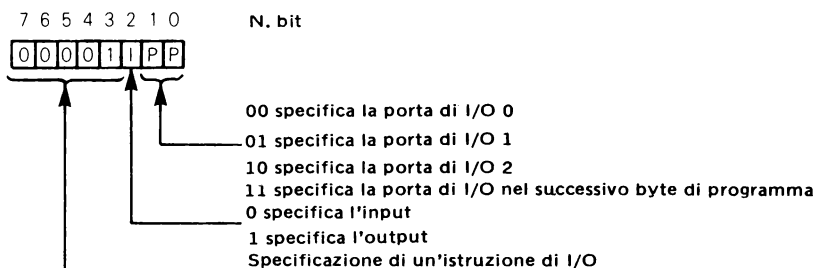
2) **Come abbiamo detto nel Capitolo 5, la maggior parte dei microcomputer hanno, o per lo meno permettono di avere, più di una porta attraverso le quali trasferire i dati fra i dispositivi esterni ed il microcomputer. Dobbiamo quindi identificare la porta di I/O attraverso la quale deve avvenire l'operazione di input o di output.**

3) **Qual'è la sorgente (per l'input) o la destinazione (per l'output), all'interno del microcomputer, per i dati che vengono trasferiti per mezzo delle istruzioni di I/O?**

Le operazioni di I/O sono così frequenti nelle applicazioni dei microcomputer, che, allo scopo di risparmiare memoria, è una buona idea includere alcune istruzioni di I/O di un solo byte.

Vorremmo usare solo quattro delle 256 scelte del codice oggetto, due per l'input (una per ogni accumulatore come destinazione), le altre due per l'output (una per ogni accumulatore come sorgente), poi specificare una delle 256 possibili porte di I/O in un byte dati che segua immediatamente.

Byte 1: Istruzione di Input o Output

Byte 2: Uso di questa porta di I/O

Per le istruzioni di I/O questo è lo schema migliore:

Abbiamo usato otto combinazioni del codice oggetto, senza specificare quale accumulatore è coinvolto nel trasferimento dei dati. Queste sono le otto combinazioni del codice oggetto usate dalle istruzioni di I/O:

00001000	Input per mezzo della porta di I/O 0
00001001	Input per mezzo della porta di I/O 1
00001010	Input per mezzo della porta di I/O 2
00001011	Input per mezzo della porta di I/O indirizzata dal byte seguente
00001100	Output per mezzo della porta di I/O 0
00001101	Output per mezzo della porta di I/O 1
00001110	Output per mezzo della porta di I/O 2
00001111	Output per mezzo della porta di I/O indirizzata dal byte seguente

Ci vorrà un bit addizionale per specificare uno dei due accumulatori in qualità di sorgente o di destinazione dei dati nelle istruzioni di I/O. Le otto combinazioni del codice oggetto precedentemente illustrate dovrebbero essere ripetute (forse con il bit 4 posizionato a 1) allo scopo di rappresentare due set di istruzioni di I/O, un set che accede all'accumulatore A0, l'altro che accede all'accumulatore A1. Come risultato, le istruzioni di I/O dovrebbero consumare 16 combinazioni del codice oggetto; e questo è un lusso inutile. **Stabiliremo invece che l'accumulatore A0 sarà sempre la sorgente o la destinazione dei dati per le istruzioni di I/O.**

ACCUMULATORE PRIMARIO

Questo uso preferenziale dell'accumulatore A0 si ritroverà spesso nel nostro set di istruzioni, dato che è un modo semplice per ridurre il numero delle combinazioni del codice oggetto usate da un qualunque tipo di istruzione. Rendendo un accumulatore più facilmente accessibile, invece di suddividere equamente le preferenze fra i due accumulatori, il programmatore può pensare a programmare in termini di accumulatore primario (A0) e accumulatore secondario (A1).

INPUT BREVE

Per le nostre istruzioni di I/O, useremo i seguenti campi codice mnemonici:

Per l'input breve:

INS P

P è l'operando dell'istruzione, e deve essere 0, 1 o 2 per specificare una delle tre porte di I/O indirizzate dalla istruzione di I/O di solo byte. L'Assembler segnalerà qualunque altro valore nel campo operando come illegale.

INPUT LUNGO

Poi, per l'input lungo:

IN P

Questa volta P può avere un qualunque valore tra 0 e 255. Un'istruzione a due byte verrà generata in questo modo:

0	0	0	0	1	0	1	1	Input in A0
								Tramite questa porta di I/O

OUTPUT BREVE

Per output breve:

OUTS P

Questa è identica all'istruzione di input breve, tranne il fatto che i dati saranno messi in output dall'accumulatore A0, attraverso la porta di I/O specificata (0, 1 o 2 soltanto).

OUTPUT LUNGO

E per l'output lungo:

OUT P

Questa istruzione è identica a quella di input lungo, tranne per il fatto che i dati saranno trasmessi dall'accumulatore A0 ad un dispositivo esterno tramite una qualunque porta di I/O tra 0 e 255.

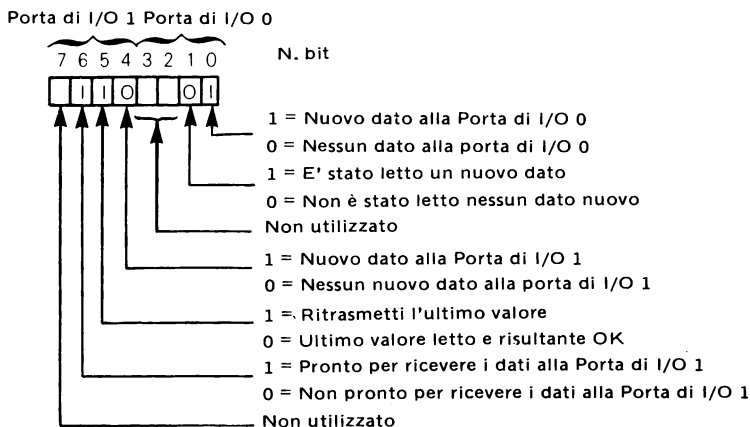
Facendo accedere le istruzioni di I/O solo all'accumulatore A0 come sorgente o destinazione di un trasferimento dati, abbiamo deciso che è più importante specificare un numero limitato di porte all'interno di un'istruzione di I/O ad un byte, che permettere che uno o l'altro dei due accumulatori sia sorgente o destinazione dei dati.

Facendo riferimento all'esempio del controllo della temperatura della doccia, supponiamo che le letture della temperatura arrivino attraverso la porta di I/O 0, mentre i controlli sono messi in output alla valvola dell'acqua calda attraverso la porta di I/O 1. La porta di I/O 2 viene usata come porta di stato comune per gli input. Le informazioni a queste porte saranno interpretate come segue:

Porta di I/O 0: Un millivoltaggio, che varia da 0 mV a 255 mV. La temperatura, in °F, si può approssimare così:

$$°F = 30 + 0{,}45 \text{ mV}$$

Porta di I/O 1: Un numero binario con segno, che specifica che la valvola dell'acqua calda deve essere aperta (positivo) o chiusa (negativo). La misura dello spostamento della valvola sarà proporzionale al valore da 0 a 127.

Porta di I/O 2: I bit di questa porta verranno interpretati come segue.

Nell'illustrazione della porta 2, 1 rappresenta i bit inseriti da un dispositivo esterno; 0 rappresenta i bit messi in output dalla CPU.

ISTRUZIONI DI RIFERIMENTO ALLA MEMORIA

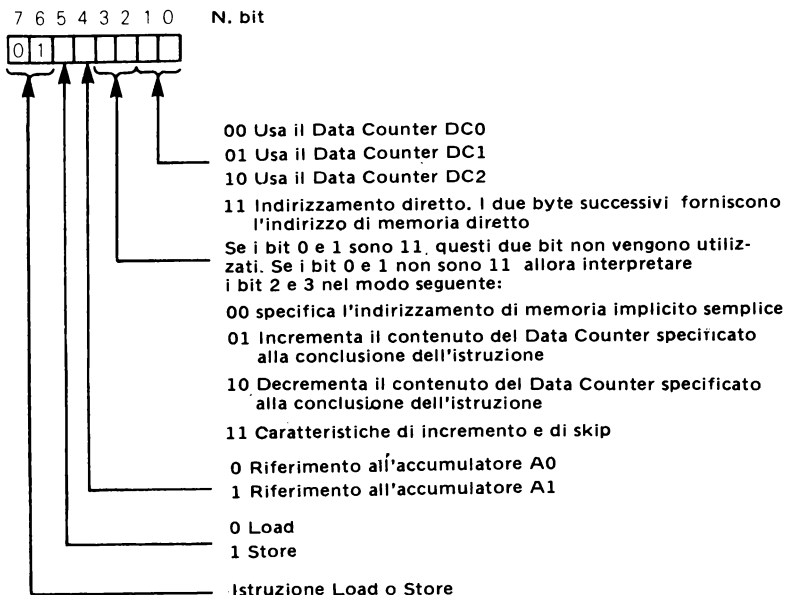
Se i dati provenienti da un lettore di temperatura arrivano in unità a più byte, ogni byte dati che viene caricato in A0 da un'istruzione di input deve essere immediatamente memorizzato nella memoria di lettura-scrittura. I dati messi in output sul controllore dell'acqua calda devono essere letti dalla memoria, caricati in A0, poi messi in output per mezzo di un'istruzione di output. L'output dei dati al controllore dell'acqua calda dipende dall'input dei dati al lettore della temperatura; nel processo di elaborazione dei dati che devono essere messi in output, ogni programma dovrà fare costantemente riferimento ai dati in memoria — caricare, memorizzare, addizionare, eseguire operazioni logiche.

L'architettura di base di un qualsiasi computer, mini o micro, ha una capacità di memorizzazione di dati, nei registri della CPU, molto limitata, e una capacità di memorizzazione dei dati (relativamente) enorme nella memoria esterna della CPU. Questo fatto rende le istruzioni di riferimento alla memoria quelle più vitali subito dopo le istruzioni di I/O. Ricordate dal Capitolo 5 che alcuni microcomputer trattano le istruzioni di I/O come un sottotest di istruzioni di riferimento alla memoria, assegnando degli indirizzi di memoria particolari alle porte di I/O.

Come ci si potrebbe aspettare, le due istruzioni di riferimento alla memoria dei microcomputer più comunemente usate spostano semplicemente i dati dà e verso la memoria; queste sono le istruzioni di Load (caricamento) e di Store (memorizzazione).

LOAD Un'istruzione di caricamento sposta i dati da una posizione di memoria ad un accumulatore.

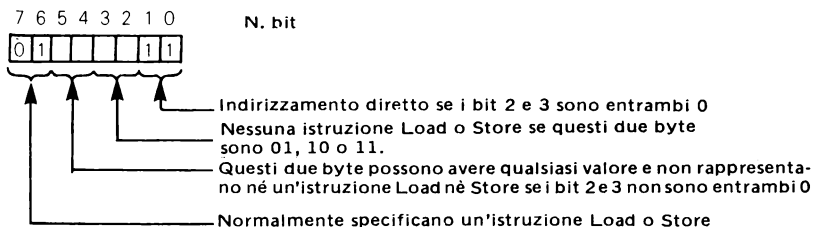
STORE Un'istruzione di memorizzazione sposta i dati da un accumulatore ad una posizione di memoria. **Essendo queste due istruzioni di riferimento alla memoria più comunemente usate, dovremo definire i bit necessari per inserire nelle istruzioni di Load e di Store dei modi d'indirizzamento molto flessibili.**

I codici oggetto delle istruzioni di Load e Store appariranno così:

Vediamo ora tutte le possibilità di indirizzamento offerte dalle istruzioni di Load e di Store, a partire dalla più semplici.

Dopo aver descritto che cosa sono i modi d'indirizzamento, li giustificheremo uno per uno.

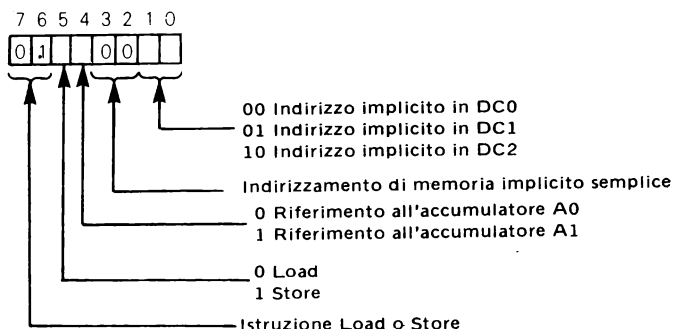
INDIRIZZAMENTO DIRETTO

Un'istruzione di Load o di Store con indirizzamento diretto avrà 1 nei bit 0 e 1, e l'indirizzo diretto verrà fornito nei due byte seguenti. Osservate che i bit 2 e 3 vengono usati per l'indirizzamento diretto; essi devono, comunque, avere un valore definito. Quindi specificheremo che i bit 2 e 3 devono essere entrambi a 0 per una istruzione di Load o di Store con indirizzamento diretto; queste istruzioni avranno ora il seguente codice oggetto:

I codici oggetto seguenti non hanno niente a che fare con le istruzioni di Load e di Store:

NUMERO DELLE ISTRUZIONI DI LOAD E DI STORE

Vi sono 64 combinazioni del codice oggetto che risultano da 01XXXXXX, dove X può essere 0 o 1; questi sono i codici oggetto dell'istruzione di Load o di Store. Dodici delle combinazioni non rappresentano le istruzioni di Load o di Store, come precedentemente illustrato (3 combinazioni dei bit 2 e 3, 4 combinazioni di tempi dei bit 4 e 5, 12 combinazioni di uguaglianza). Perciò, vi sono 52 combinazioni delle istruzioni di Load e di Store.

INDIRIZZAMENTO IMPLICITO

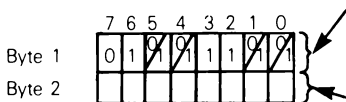
Le istruzioni di Load e di Store con indirizzamento in memoria implicito possono avere uno qualunque dei seguenti codici oggetto:

L'indirizzo di memoria effettivo per l'istruzione di Load o di Store è il contenuto del Data Counter DC0, DC1 o DC2, a seconda di quale sia stato specificato dai bit 0 e 1.

AUTOINCREMENTO - AUTODECREMENTO

La presentazione dell'autoincremento e dell'autodecremento è molto facile da capire; la caratteristica dell'autoincremento è che l'indirizzo di memoria implicito, cioè il contenuto del Data Counter specificato, sarà incrementato di 1 alla conclusione dell'istruzione di riferimento alla memoria. Viceversa, la caratteristica dell'autodecremento è che il contenuto del Data Counter sarà decrementato di 1 alla conclusione dell'istruzione.

Il codice oggetto per un'istruzione di autoincremento sarà diverso dal codice oggetto equivalente dell'indirizzamento implicito solo nei bit 3 e 2, che saranno 01; l'istruzione di autodecremento equivalente avrà 10 nei bit 3 e 2.

INCREMENTO E SKIP

Incremento e skip, specificato dal fatto di aver 1 nei bit 2 e 3, non è una caratteristica comune dei microcomputer; il PDP-8 della Rockwell è un microcomputer descritto nel Volume 2 che ha queste caratteristiche. Creeremo ora istruzioni di Load e di Store con il seguente formato:

Load o Store, poi incrementare il contenuto del Data Counter specificato dai bit 0 e 1; poi controllare il contenuto del Data Counter.

Se gli ultimi 6 bit del contenuto del Data Counter sono 000000, incrementare il Program Counter per andare oltre questo byte ignorandolo. Se gli ultimi 6 bit del Data Counter hanno un altro valore trattare questo byte come un numero binario con segno, da aggiungere al contenuto del Program Counter, forzando un salto.

Il modo più efficace per illustrare la necessità dei vari modi d'indirizzamento è con sequenze di programma brevi. Descriveremo quindi prima i campi codice mnemonico usati per le istruzioni di Load e di Store.

LOAD DIRETTO

STORE DIRETTO

Load e Store diretto useranno questi campi codice mnemonico:

LRA	ADDR	Caricare direttamente in A0
LRB	ADDR	Caricare direttamente in A1
SRA	ADDR	Memorizzare direttamente da A0
SRB	ADDR	Memorizzare direttamente da A1

ADDR è un qualunque simbolo che rappresenti una posizione di memoria dalla quale verranno letti i dati o nella quale i dati verranno scritti. Usiamo la lettera A per rappresentare A0 e B per rappresentare A1; potremmo usare le cifre 0 e 1, ma è troppo facile confondere lo zero con la o, e l'uno con la i,; per questo motivo l'uso di 0 e 1 nei campi codice mnemonico delle istruzioni non è per niente comune.

LOAD IMPLICITO

STORE IMPLICITO

Load e Store implicito useranno questi campi codice mnemonico:

LMA	DCX	Caricare in A0 dalla posizione di memoria indirizzata da DCX
LMB	DCX	Caricare in A1 dalla posizione di memoria indirizzata da DCX
SMA	DCX	Memorizzare il contenuto di A0 nella posizione di memoria indirizzata da DCX
SMB	DCX	Memorizzare il contenuto di A1 nella posizione di memoria indirizzata da DCX

DCX specifica uno dei tre Data Counter e perciò deve essere DC0, DC1 o DC2.

LOAD/STORE CON AUTOINCREMENTO O AUTODECREMENTO

L'istruzione di Load o Store con autodecremento o autoincremento sarà identica a Load/Store implicito, come descritto in precedenza, tranne per il fatto che il contenuto del Data Counter specificato sarà incrementato o decrementato. Useremo i campi codici mnemonico LNA e

LNB per Load con autoincremento, LDA e LDB per Load con autodecremento, SNA e SNB per Store con autoincremento e SDA e SDB per Store con autodecremento.

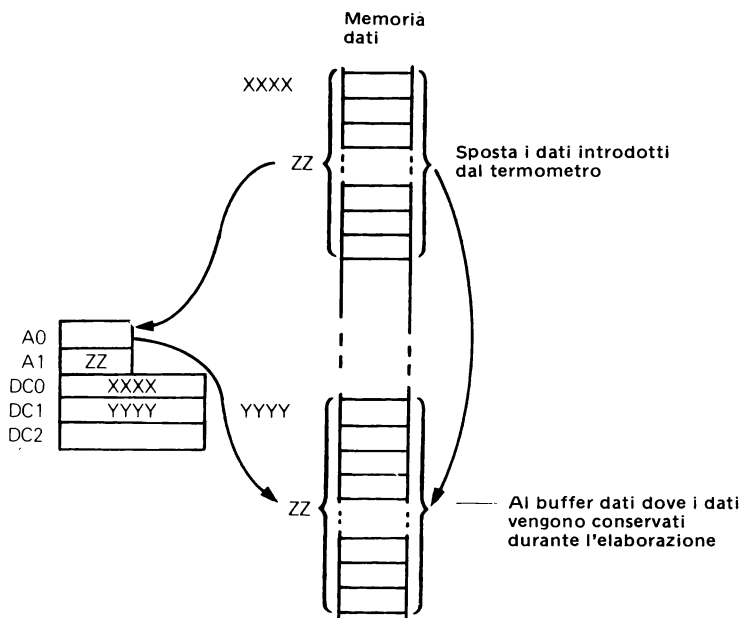
LOAD/STORE CON AUTOINCREMENTO E SKIP

Le istruzioni di Load e Store con autoincremento e Skip saranno specificate dai seguenti campi codice mnemonico:

LSA	DCX,LABEL	Caricare in A0
LSB	DCX,LABEL	Caricare in A1
SSA	DCX,LABEL	Memorizzare il contenuto di A0
SSB	DCX,LABEL	Memorizzare il contenuto di A1

DCX identifica il Data Counter che contiene l'indirizzo di memoria implicito; perciò deve essere DC0, DC1 o DC2.

LABEL è un simbolo che identifica l'istruzione che sarà eseguita successivamente se, dopo che DCX è stato incrementato, le ultime sei cifre binarie non sono tutte a zero.

Allo scopo di dimostrare quello che possono fare le istruzioni di Load e di Store, vediamo un semplice programma che sposta dei dati da un buffer all'altro. Supponendo che gli indirizzi di partenza dei buffer sorgente e destinazione siano nei Data Counter DC0 e DC1, e supponendo che la lunghezza dei buffer sia memorizzata nell'accumulatore A1, il problema può essere illustrato in questo modo:

XXXX è l'indirizzo iniziale del buffer dati in input
YYYY è l'indirizzo iniziale del buffer dati in output
ZZ è la lunghezza dei buffer dati

La seguente sequenza di istruzioni eseguirà lo spostamento di dati richiesto:

```
LOOP    LMA     DC0     Carica il byte dati in input seguente
        SMA     DC1     Memorizza nel byte del buffer destinazione seguente
        Incrementare il contenuto di DC0
        Incrementare il contenuto di DC1
        Decrementare il contenuto di A1
        Se A1 contiene 0, salta a LOOP
```

Le istruzioni che non abbiamo ancora descritto sono scritte a parole, anzichè con i campi codice mnemonico, con cui non si ha molta dimestichezza.

GIUSTIFICAZIONE DI AUTOINCREMENTO O AUTODECREMENTO

Usiamo ora la possibilità di autoincremento, ed ecco che cosa succede alla nostra sequenza di istruzioni:

```
LOOP    LNA     DC0     Carica il byte dati in input successivo.
                        Incrementa l'indirizzo.
        SNA     DC1     Memorizzare nel successivo byte del buffer destina-
                        zione.
        Decrementa il contenuto di A1
        Se A1 contiene 0, salta a LOOP
```

Da una sequenza di sei istruzioni, sono state tolte due istruzioni e sono stati risparmiati due byte di codice oggetto.

GIUSTIFICAZIONE DI AUTOINCREMENTO E SKIP

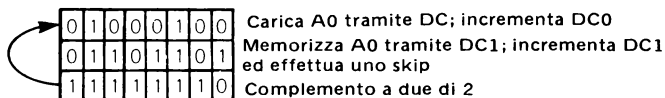
Ora supponiamo che il buffer di destinazione finisca alla posizione di memoria $08C0_{16}$: le ultime sei cifre binarie di questo indirizzo sono tutte a zero:

$$08C0_{16} = 0000100011\underbrace{000000}_{\text{Controllate per l'autoskip}}$$

Possiamo ora ridurre il nostro programma di spostamento dati a queste due istruzioni:

```
LOOP    LNA     DC0        Carica il successivo byte dati in input; incrementa l'in-
                           dirizzo.
        SSA     DC1,LOOP   Memorizza, incrementa ed effettua uno skip alla fine.
```

Queste due istruzioni occupano tre byte, in questo modo:

0	1	0	0	0	1	0	0	**Carica A0 tramite DC; incrementa DC0**
0	1	1	0	1	1	0	1	**Memorizza A0 tramite DC1; incrementa DC1 ed effettua uno skip**
1	1	1	1	1	1	1	0	**Complemento a due di 2**

Non abbiamo più bisogno di tenere la lunghezza del buffer nell'accumulatore A1. Non abbiamo nemmeno bisogno di decrementare la lunghezza del buffer, o incrementare gli indirizzi di memoria. Dopo aver incrementato l'indirizzo del buffer destinazione, l'istruzione di Store, incremento e Skip controlla il valore incrementato; se quest'ultimo non finisce con sei cifre binarie a zero, l'esecuzione ritornerà all'istruzione di Load; questo loop di due istruzioni verrà continuamente rieseguito fino a che l'istruzione di Store, Incremento e Skip fa andare l'indirizzo di destinazione a $08C0_{16}$; a questo punto il salto non verrà effettuato e verrà eseguita l'istruzione che viene immediatamente dopo il loop dello spostamento dati.

Il programmatore di un minicomputer resterebbe perplesso davanti ad uno schema di indirizzamento quale l'autoincremento e skip. Il fatto che le tabelle dati debbano essere messe a indirizzi di memoria che finiscono con sei zeri binari presenta più problemi che vantaggi.

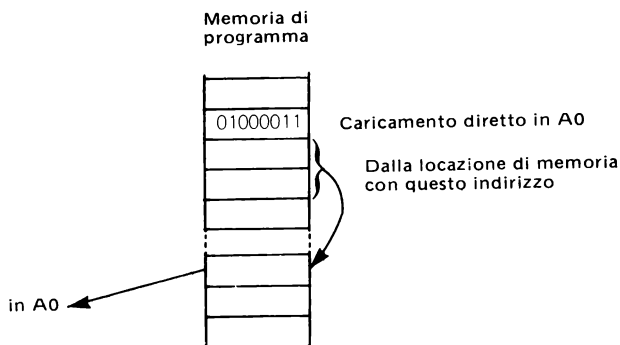
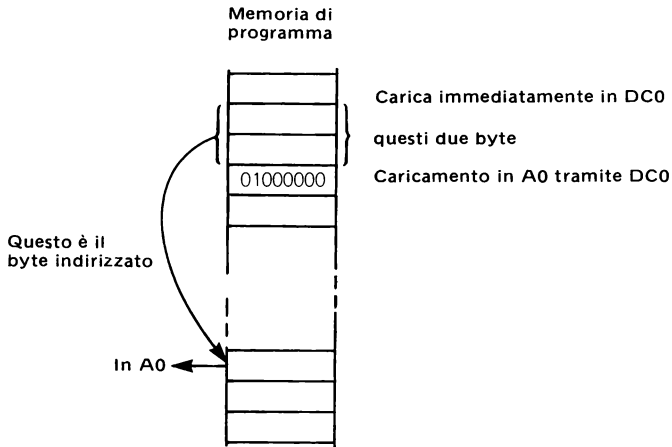
Anche se il programmatore di minicomputer può ammirare la bellezza dei loop che non richiedono una logica speciale di fine loop, i problemi associati con il riposizionamento dei dati sarebbero terribili; se un programma dovesse essere riusato in un'altra applicazione, o se fosse parte di un sistema a divisione di tempo, il programmatore dovrebbe costantemente preoccuparsi di essere sicuro che le tabelle dati finiscano in punti precisi della memoria — oppure lasciar perdere tutto.

Ritorniamo ancora una volta al vecchio detto del programmatore di minicomputer: "Ricordate che qualunque cosa facciate oggi, può darvi fastidio domani". Ma ricordate che nel mondo dei microcomputer non esiste domani. Qualunque cosa facciate oggi diventa un chip della ROM e non cambierà mai. La suddivisione delle tabelle dati sui confini degli indirizzi di memoria è solo un inconveniente secondario, dato che la suddivisione della memoria sarà comunque una parte significativa della programmazione dei microcomputer. Quando i chip ROM che contengono il vostro programma possono essere riprodotti migliaia di volte, dovete essere assolutamente certi che il vostro programma stia nel chip più ridotto possibile.

La caratteristica di autoincremento e skip offre dei vantaggi molto interessanti nella applicazione di un microcomputer, perchè risparmia sui byte del programma oggetto, mentre quello che ci si rimette — e cioè dover suddividere le tabelle dati in modo che cadano sui confini d'indirizzo — è parte di un compito che deve essere eseguito in qualunque caso.

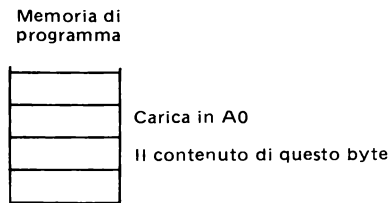
GIUSTIFICAZIONE DELL'INDIRIZZAMENTO DIRETTO

Non abbiamo ancora giustificato la necessità delle istruzioni d'indirizzamento diretto. Sono veramente necessarie?

Non c'è niente di quello che fa un'istruzione d'indirizzamento diretto che non potrebbe essere fatto con un'istruzione d'indirizzamento implicito; in certi casi, comunque, le istruzioni d'indirizzamento diretto usano meno memoria. Consideriamo la lunghezza del buffer che stiamo per caricare nell'accumulatore A1, e che poi dobbiamo decrementare. Alla fine abbiamo eliminato questa sequenza logica dal nostro esempio di spostamento dati, ma vi saranno molti esempi in cui questo tipo di logica non può essere eliminata. Come si fa a caricare nell'accumulatore la lunghezza del buffer, o qualunque numero simile? Ecco come eseguirà tale compito un'istruzione d'indirizzamento diretto a tre byte:

L'uso dell'indirizzamento di memoria implicito richiederà quattro byte per l'operazione, che inoltre userà temporaneamente un Data Counter, così:

L'istruzione immediata a tre byte che è necessaria per caricare i dati nel Data Counter è una spesa non necessaria, quando è seguita da un solo riferimento alla memoria, come il caricamento in una volta di un indice in un altro registro; come abbiamo visto in precedenza in questo Capitolo, i tre byte necessari per caricare un indirizzo in un Data Counter permettono in seguito un grosso risparmio di memoria, ma solo se l'indirizzo del Data Counter verrà usato più volte.

Se un microcomputer deve avere solo l'indirizzamento diretto o implicito, l'indirizzamento implicito è il migliore; ad esempio, l'Intel 8008, che è stato il predecessore dell'Intel 8080, ha indirizzamento implicito ma non quello diretto.

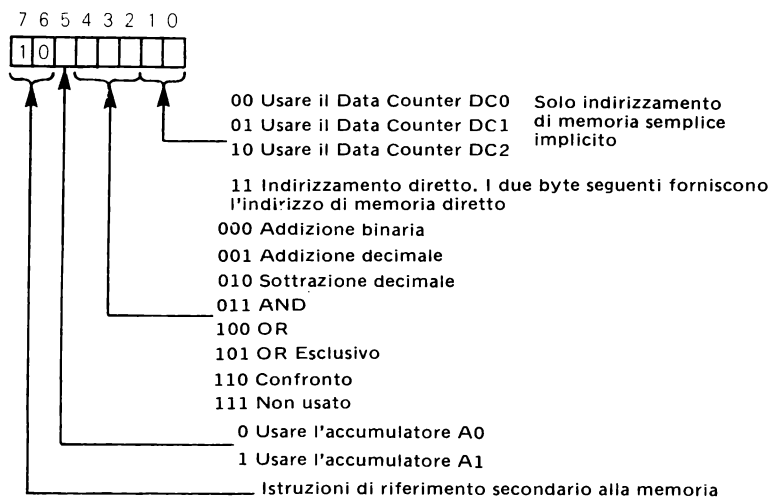
La maggior parte dei programmi caricano dei singoli valori (come contatori e indici) nei registri abbastanza spesso da rendere l'indirizzamento diretto giustificabile.

Notate che se il contatore o l'indice che deve essere caricato in un registro ha un valore che non cambierà mai, non userete nè l'indirizzamento diretto nè quello implicito per caricare il valore in un registro. Userete l'indirizzamento immediato:

ISTRUZIONI DI RIFERIMENTO SECONDARIO ALLA MEMORIA (OPERAZIONI DI RIFERIMENTO ALLA MEMORIA)

Vediamo ora le istruzioni di riferimento alla memoria diverse da Load e Store. In ogni caso sarà eseguita un'operazione che fa uso del contenuto di uno degli accumulatori, oltre al contenuto di una posizione di memoria indirizzata. Il risultato sarà

sempre memorizzato nell'accumulatore identificato, cancellando il valore precedentemente contenuto (l'istruzione di confronto descritta più avanti è un'eccezione). I flag di stato Zero, Segno, Carry e Overflow verranno settati o risettati per riflettere il risultato dell'operazione. Per esempio, l'istruzione "Addiziona la memoria ad A1" addizionerà il contenuto della posizione di memoria indirizzata al contenuto dello accumulatore A1. Il contenuto precedente della memoria non viene cambiato.

Ad eccezione dell'istruzione di Store, le istruzioni del microcomputer eviteranno di modificare la memoria dato che ciò implica la presenza di una memoria di lettura-scrittura.

Inseriamo queste istruzioni di rifermento secondario alla memoria:

Ecco i campi codice mnemonico delle istruzioni:

ABA	o	ABB	Addizione binaria a A0 o A1
ADA	o	ADB	Addizione decimale ad A0 o A1
DSA	o	DSB	Sottrazione decimale da A0 o A1
ANA	o	ANB	AND con A0 o A1
ORA	o	ORB	OR con A0 o A1
XRA	o	XRB	OR esclusivo con A0 o A1
CMA	o	CMB	Confronto di A0 o A1 con la memoria

Il codice "10" nei bit 7 e 6 specifica che i sei bit rimanenti rappresentano istruzioni di riferimento secondario alla memoria. Comunque, solo sette delle otto combinazioni possibili per i bit 2, 3 e 4 vengono usate per queste istruzioni. Perciò, solo 56 delle

64 combinazioni di bit sono, di fatto, istruzioni di riferimento secondario alla memoria.

In ogni caso, l'istruzione sarà descritta in questo modo:

MNEM DCX

dove MNEM è uno dei campi codice mnemonico elencati prima (es. ADA) e DCX è uno dei Data Counter (DC0, DC1 o DC2).

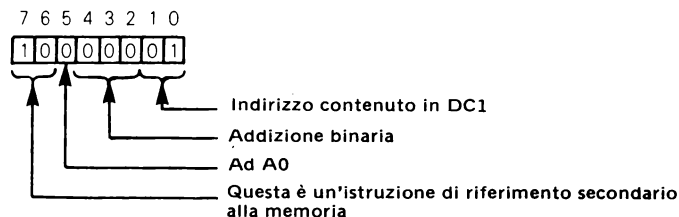
La versione con riferimento diretto alla memoria dell'istruzione si presenterà così:

MNEM ADDR

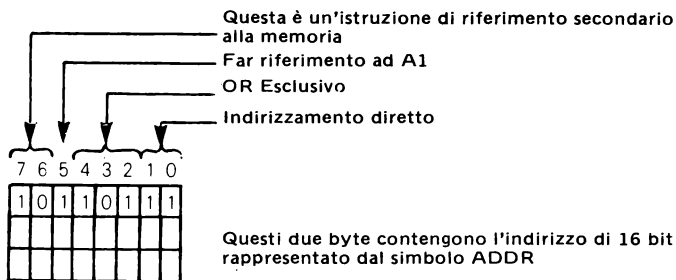
dove ADDR è l'indirizzo diretto.

Ecco due esempi. L'istruzione:

ABA DC1

addiziona ad A0, usando l'addizione binaria, il contenuto della posizione di memoria il cui indirizzo è contenuto in DC1. Questo è il codice oggetto generato:

Quella che segue è un'istruzione di riferimento diretto alla memoria:

XRB ADDR

Sul contenuto di A1, viene effettuata un'operazione di OR Esclusivo con il contenuto della posizione di memoria indirizzata dal simbolo ADDR. Verranno creati questi tre byte di programma oggetto:

Solo due delle sette istruzioni di riferimento secondario alla memoria che abbiamo descritto hanno bisogno di qualche commento.

"Add e subtract decimal" eseguono addizioni o sottrazioni decimali, usando tre o due passi binari, come abbiamo descritto per l'aritmetica in binario decimale codificato nel Capitolo 2.

Eseguiamo le sottrazioni decimali usando un'istruzione diversa, dato che la sequenza logica è sufficientemente diversa da un'addizione decimale da rendere vantaggioso avere un'istruzione in più.

SOTTRAZIONE BINARIA

ADATTAMENTO DECIMALE

Non forniamo un'istruzione di istruzione di sottrazione binaria separata, dato che questa è semplicemente il complemento a due seguito da un'addizione, come descritto nel Capitolo 2. Notate attentamente che le istruzioni di addizione e sottrazione decimale e l'istruzione di adattamento decimale non sono la stessa cosa. L'adattamento decimale prende semplicemente il contenuto di un registro e risistema i bit per creare l'equivalente decimale del numero binario; questo è spiegato nel Capitolo 2. La maggior parte dei microcomputer ha l'istruzione di adattamento decimale; una piccola parte ha le istruzioni di aritmetica decimali.

CONFRONTO

L'istruzione di confronto sottrae il contenuto della posizione di memoria indirizzato dall'accumulatore specificato; il risultato dell'operazione viene perso — non viene memorizzato nell'accumulatore specificato. Questa è un'istruzione molto utile, dato che permette alla sequenza dell'esecuzione del programma di essere condizionata dalla grandezza relativa dei dati.

Le istruzioni di salto condizionato, descritte più avanti in questo Capitolo, fanno uso delle istruzioni di confronto, e vengono usate insieme a queste ultime.e.

GIUSTIFICAZIONE DELLE ISTRUZIONI DI RIFERIMENTO SECONDARIO ALLA MEMORIA

Le istruzioni di riferimento secondario alla memoria sono necessarie? A questa domanda, si può rispondere in due modi:

Primo, le operazioni eseguite dalle istruzioni di riferimento secondario alla memoria sono necessarie.

Secondo, queste operazioni devono essere eseguite usando le istruzioni di riferimento secondario alla memoria.

Le operazioni descritte — addizione, logica Booleana e confronto sono punti così fondamentali in qualunque sequenza logica che un microcomputer che non offrisse, in un modo o nell'altro, queste possibilità logiche, non avrebbe alcun valore.

Comunque, non vi è nessuna ragione per cui queste istruzioni debbano essere parte delle istruzioni di riferimento alla memoria. Per esempio, sarebbe possibile caricare i due operandi negli accumulatori A0 e A1, poi eseguire le stesse operazioni da registro a registro. I microcomputer che hanno molti accumulatori, come l'Intel 8080, favoriscono le istruzioni tra registro e registro rispetto a quelle registro-memoria; i microcomputer con meno accumulatori, come il Motorola M6800, usano le istruzioni registro-memoria, come le abbiamo descritte.

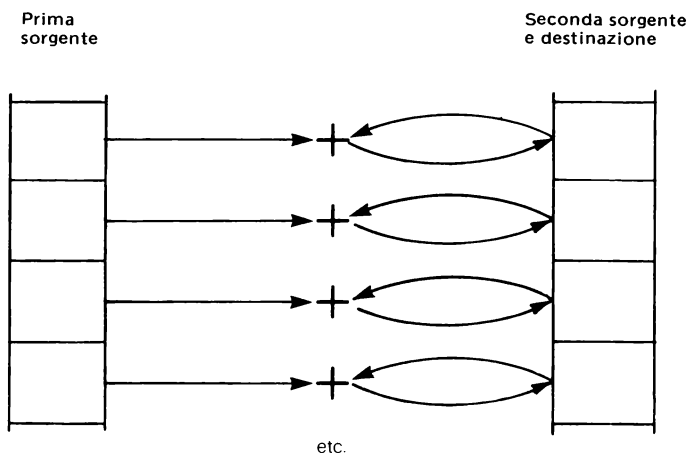
Vediamo un esempio facile.

ADDIZIONE A PIU' BYTE

Supponiamo che i due buffer che iniziano a XXXX e YYYY (nell'illustrazione precedente) contengano ognuno un solo numero a più byte. Il numero nel buffer che inizia a XXXX potrebbe essere aggiunto al numero nel buffer che inizia a YYYY, in questo modo:

```
             Inizialmente azzerare lo stato Carry
LOOP   LMA    DC0         Carica il byte in input seguente
       ABA    DC1         Addizione binarie dal buffer del risultato
       SSA    DC1,LOOP    Memorizza il risultato, incrementa ed effet-
                          tua uno skip
```

Il suddetto programma di tre istruzioni presuppone che il buffer che inizia a YYYY

(questo indirizzo è memorizzato in DC1) contenga uno dei numeri che devono essere sommati; ma alla fine dell'addizione questo buffer conterrà il risultato. Questa logica funziona, dato che il risultato viene memorizzato sul byte che è appena stato sommato; perciò, l'informazione viene distrutta solo quando non è più necessaria. Ecco come possiamo illustrare questo concetto:

Un altro loop di tre istruzioni può eseguire l'addizione binaria il cui risultato viene memorizzato in un terzo buffer, che supporremo sia indirizzato da DC2. Le tre istruzioni appaiono in questo modo:

```
LOOP  LNA   DC0        Carica il byte seguente all'augendo
      ABA   DC1        Addiziona il byte dell'addendo corrispondente
      SSA   DC2,LOOP   Memorizza il risultato, incrementa ed effettua uno skip.
```

GIUSTIFICAZIONE DELLA LOGICA BOOLEANA: CONTROLLO DEL CAMBIAMENTO DEGLI SWITCH

Un esempio dell'utilità delle istruzioni secondarie di riferimento alla memoria sta nel controllare il cambiamento del valore degli switch.

Supponiamo che lo stato di otto switch sia preso in input alla porta I/O 4; le precedenti posizioni di questi otto switch sono memorizzate nella posizione di memoria indirizzata dal simbolo SWITCH. La seguente sequenza di istruzioni identifica quali switch hanno cambiato valore e in che modo:

```
IN    4        Prende in input i valori degli switch
XRA   SWITCH   Identifica gli switch cambiati
Conserva il contenuto di A0 in A1
ANA   SWITCH   Identifica quali switch sono passati da 0 a 1
```

Ecco che cosa fanno le tre istruzioni suddette:

La prima istruzione prende in input i nuovi valori degli switch in A0; supponiamo che i valori siano:

01100101

Dove 0 rappresenta uno switch "chiuso" e 1 rappresenta uno switch "aperto".

Supponiamo che i precedenti valori degli switch, memorizzati nella posizione di memoria identificata dal simbolo SWITCH, fossero:

10101101

Gli switch 7 e 3 erano "aperti" ed ora sono "chiusi". Lo switch 6 era "chiuso" ed ora è "aperto". Gli switch 5, 4, 2, 1 e 0 non sono cambiati.

OR ESCLUSIVO

L'istruzione XRA esegue l'OR Esclusivo tra la situazione vecchia e nuova degli switch:

Vecchia situazione:	10101101	
Nuova situazione:	01100101	
XRA:	11001000	**dà gli switch cambiati**

AND

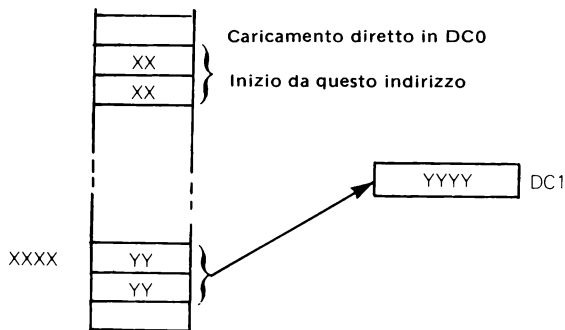
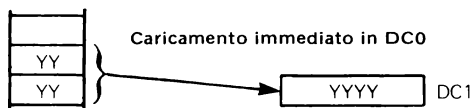
L'istruzione di AND esegue l'AND della vecchia situazione con la configurazione degli switch cambiati:

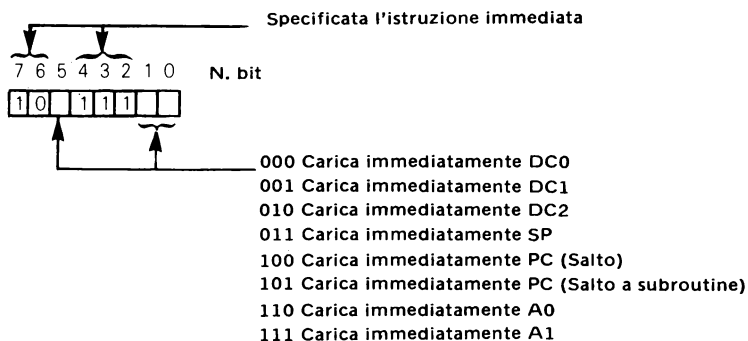
Situazione cambiata:	11001000	
Vecchia situazione:	10101101	
ANA:	10001000	**dà gli switch passati da 0 a 1**

ISTRUZIONI DI CARICAMENTO IMMEDIATO, SALTO E SALTO AD UNA SUBROUTINE

Il concetto di indirizzamento immediato è stato usato spesso come mezzo per caricare dati o indirizzi nei registri. Fino a che punto sono vitali per il set di istruzioni di un microcomputer le istruzioni d'indirizzamento immediato?

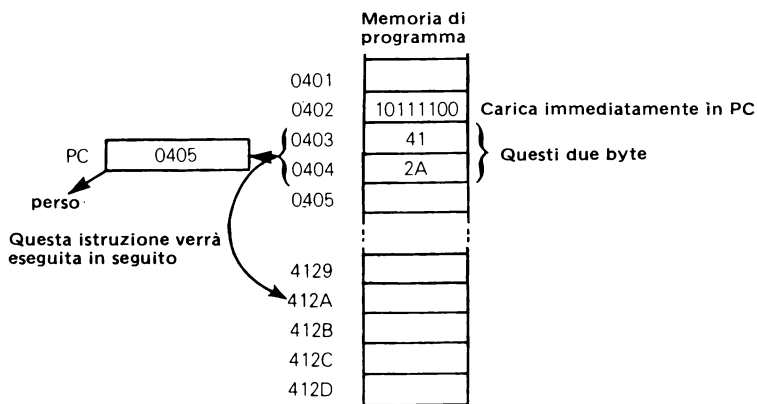
GIUSTIFICAZIONE DELLE ISTRUZIONI IMMEDIATE

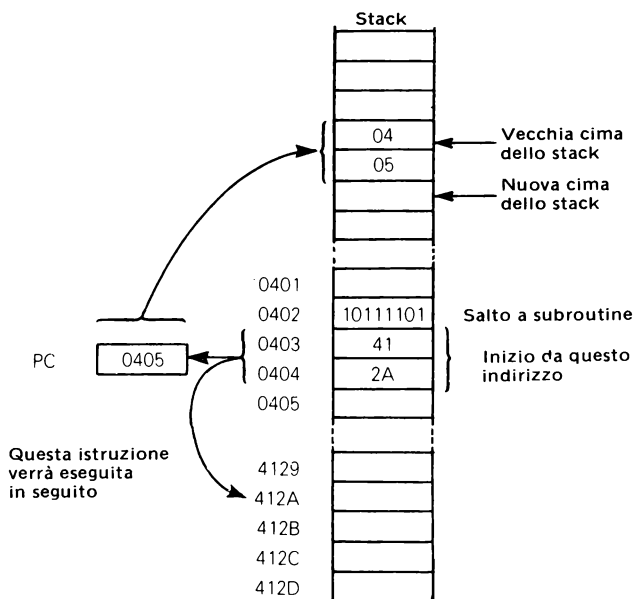
Non possiamo usare l'indirizzamento in memoria implicito per caricare un indirizzo in un Data Counter, dato che l'indirizzamento in memoria implicito richiede che il Data Counter contenga già un indirizzo. A questo scopo, si potrebbe usare l'indirizzamento diretto. Si potrebbe indirizzare in modo diretto un indirizzo base memorizzato in due byte di memoria, e caricarlo in un Data Counter, così:

Ma l'illustrazione precedente ha chiaramente dei byte ridondanti; l'indirizzo che viene caricato nel Data Counter potrebbe altrettanto facilmente essere memorizzato in due byte d'indirizzo di memoria diretto, così:

Le istruzioni immediate non sono assolutamente vitali per il set di istruzioni di un microcomputer, ma rappresentano sicuramente una grossa convenienza; perciò includeremo otto istruzioni d'indirizzamento immediato: per caricare dati nei tre Data Counter, nello Stack Pointer, nel Program Counter (con due variazioni) o nei due accumulatori.

Queste istruzioni saranno a due o a tre byte; dato che gli accumulatori sono lunghi solo un byte, le istruzioni immediate che caricano i dati in un accumulatore saranno seguite da un solo byte di dati immediati. Il Data Counter, il Program Counter e lo Stack Pointer sono tutti lunghi due byte; perciò, le istruzioni immediate che caricano i dati in uno di questi registri saranno seguiti da due byte di dati immediati. **La seguente configurazione di codice oggetto specificherà le istruzioni immediate:**

Dato che vi sono otto istruzioni immediate, e dato che vi erano otto combinazioni di codici oggetto non usate dall'interno del gruppo di istruzioni di riferimento secondario alla memoria, usiamo queste otto combinazioni libere per le istruzioni immediate, come illustrato in precedenza.

ISTRUZIONI DI SALTO

Dobbiamo prestare particolare attenzione alle due istruzioni che caricano in immediato il Program Counter; a differenza delle altre istruzioni immediate, queste due modificano la sequenza di esecuzione del programma; l'istruzione eseguita successivamente verrà prelevata dalla posizione di memoria il cui indirizzo è stato caricato in modo immediato nel Program Counter.

Questa è un'istruzione di salto (JUMP o BRANCH) incondizionato nella sua forma più semplice:

ISTRUZIONE DI SALTO AD UNA SUBROUTINE

Un'istruzione di salto ad una subroutine si differenzia solo per il fatto che il contenuto attuale del Program Counter deve essere messo da parte prima che i nuovi dati vengano caricati in modo immediato nel Program Counter; dato che il nostro microcomputer ha uno stack, il contenuto del Program Counter verrà messo nello stack (come già descritto in questo capitolo).

La maggior parte delle descrizioni dei set di istruzioni per microcomputer non mettono le istruzioni di salto e di salto ad una subroutine nella categoria delle istruzioni immediate; comunque, la logica di questo tipo di istruzione è quasi del tutto identica a quella delle istruzioni di tipo immediato.

CARICAMENTO IN MODO IMMEDIATO

I campi codice mnemonico delle istruzioni immediate saranno diversi per le istruzioni di salto e per quelle di caricamento diretto. **Per il caricamento immediato, useremo i seguenti campi mnemonici:**

```
LIM    R,DATA
```

R deve essere A0, A1, DC0, DC1, DC2 o SP. DATA deve essere un numero o un simbolo che rappresenti un numero; deve essere equivalente ad un valore di 8 bit se R è A0 o A1; deve essere equivalente ad un valore di 16 bit in caso diverso.

SALTO (JUMP)

L'istruzione di salto apparirà in questo modo:

```
JMP    ADDR
```

ADDR deve essere l'etichetta dell'istruzione che deve essere eseguita dopo.

SALTO AD UNA SUBROUTINE

L'istruzione di salto ad una subroutine apparirà così:

```
JSR    SNAME
```

SNAME deve essere l'etichetta della prima istruzione eseguibile all'intero della subroutine.

Creeremo ora una subroutine.

SUBROUTINE

Ritorniamo al programma di spostamento dati che illustrava l'istruzione di Incremento e Skip; scritto per intero, questo programma dovrebbe comprendere addizionali per caricare gli indirizzi nei Data Counter, così:

```
BUFA   EQU    XXXX
BUFB   EQU    YYYY
       ___
       ___
       ___
       LIM    DC0,BUFA      Carica l'indirizzo iniziale sorgente
       LIM    DC1,BUFB      Carica l'indirizzo iniziale di destinazione
LOOP   LNA    DC0           Sposta i dati dalla sorgente
       SSA    DC1,LOOP      Verso la destinazione
```

DIRETTIVE DI ASSEGNAZIONE ASSEMBLER (EQUATE)

I campi codice mnemonico EQU rappresentano le direttive di assegnazione dell'Assembler. Ricordate che una direttiva di Assembler non è un'istruzione, e non genera nessun codice oggetto; fornisce invece all'Assembler informazioni senza le quali l'Assembler non potrebbe generare il programma oggetto.

EQU identifica una direttiva d'assegnazione; questa direttiva dice all'Assembler che, in qualunque punto del campo etichetta, appaia un certo simbolo, il numero nel campo operando deve essere sostituito. Per esempio, dice all'Assembler:

"Usa il valore esadecimale XXXX in qualunque punto tu veda il simbolo BUFA".

A questo punto potremmo riscrivere il nostro programma in questo modo:

```
        LIM   DC0,XXXX
        LIM   DC1,YYYY
LOOP    LNA   DC0
        SSA   DC1,LOOP
```

DIRETTIVE DI ASSEMBLER — LORO VALORE

Il vantaggio di avere l'assegnazione è che il simbolo BUFA (O BUFF) può apparire molte volte all'interno di uno stesso programma. Se il valore associato con il simbolo cambia, tutto quello che dovete fare è cambiare un'assegnazione nel programma sorgente. Quando riassemblate il programma sorgente, ogni riferimento al simbolo cambiato sarà sostituito correttamente nel nuovo programma oggetto creato dall'Assembler.

Senza la direttiva di assegnazione, dovreste trovare tutte le istruzioni del programma sorgente che si riferiscono al simbolo cambiato, poi dovreste correggerle tutte, senza nessuna garanzia di averle trovate tutte.

SALTO AD UNA SUBROUTINE

Per trasformare il programma di spostamento dati in una subroutine, tutto ciò che facciamo è dare una etichetta all'istruzione che deve essere eseguita per prima, ed aggiungere un'istruzione che esegua un rientro della subroutine:

```
MOVE    LIM   DC0,BUFA    Carica l'indirizzo iniziale della sorgente
        LIM   DC1,BUFB    Carica l'indirizzo iniziale della destinazione
LOOP    LNA   DC0         Sposta i dati dalla sorgente
        SSA   DC1,LOOP    Destinazione
        Rientra dalla subroutine
```

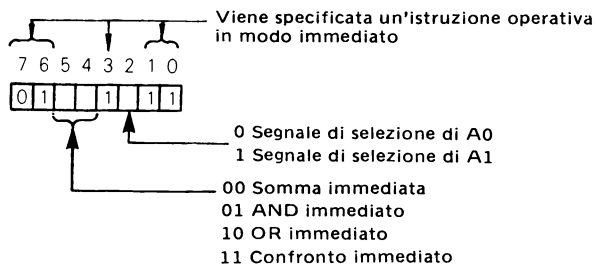
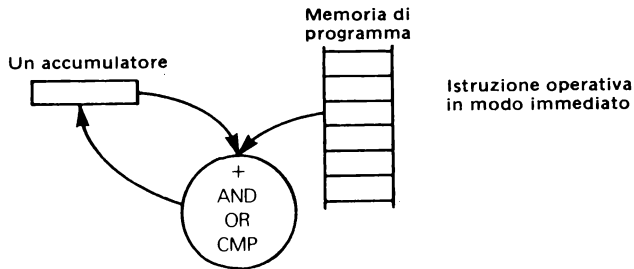
L'istruzione di rientro dalla subroutine è descritta con le istruzioni dello stack; per ora ignoreremo la logica del rientro.

I nostri programmi possono chiamare la subroutine con questa istruzione:

```
JSR    MOVE
```

La subroutine MOVE può essere chiamata da qualunque altro programma, un numero qualsiasi di volte.

ISTRUZIONI OPERATIVE IN MODO IMMEDIATO

Un numero limitato di istruzioni operative in modo immediato sarà molto utile; queste istruzioni eseguono operazioni sul contenuto di un accumulatore con l'operando immediato, memorizzando il risultato nell'accumulatore specificato. Consideriamo questa istruzione:

Ogni istruzione descrive un'operazione che sarà eseguita usando il contenuto di un accumulatore e il byte che segue il codice istruzione:

I flag di stato, C, O e Z, verranno settati o risettati per riflettere il risultato della operazione.

Osservate che abbiamo usato otto delle dodici combinazioni del codice oggetto non usate dalle 64 combinazioni di Load/Store 01XXXXXX. Queste quattro combinazioni continuano a non essere usate all'interno di questa configurazione:

01000111
01010111
01100111
01110111

Queste quattro combinazioni possono essere rappresentate da 01XX0111.

ADDIZIONE IMMEDIATA
AND IMMEDIATO
OR IMMEDIATO
CONFRONTO IMMEDIATO

Useremo i seguenti campi codice mnemonico per le istruzioni operative in modo immediato:

AIA	DATA	Somma immediata in A0
AIB	DATA	Somma immediata in A1
NIA	DATA	AND immediato in A0
NIB	DATA	AND immediato in A1
OIA	DATA	OR immediato con A0
OIB	DATA	OR immediato con A1
CIA	DATA	OR esclusivo immediato con A0
CIB	DATA	OR esclusivo immediato con A1

In ogni caso DATA è un numero (o un simbolo) che diventa un valore di 8 bit e, in ogni caso, verranno generati due byte di codice oggetto. Per esempio, l'istruzione OIB creerà questo codice oggetto:

Byte 1	0	1	1	0	1	1	1	1	**OR immediato con A1**
Byte 2									**il dato in questo byte**

GIUSTIFICAZIONE DELLE ISTRUZIONI OPERATIVE IN MODO IMMEDIATO

Dimostreremo ora il valore delle istruzioni operative in modo immediato. Guardiamo ancora come, nella descrizione dell'istruzione di I/O la porta 2 veniva definita come controllo della combinazione e porta di stato.

Queste due istruzioni determinano se vi sono nuovi dati alla porta di I/O:

INS	2	Prende in input lo stato
NIA	H'01'	Maschera tutto tranne il bit 0

H'01' significa 01 esadecimale.

L'istruzione NIA risetta a 0 tutti i bit dell'accumulatore diversi dal bit 0:

		Contenuto di A0
INS	2	XXXXXXXX
NIA	H'01'	0000000X

X rappresenta o 0 o 1.

Se il risultato è zero, vuol dire che il bit 0 era a Zero, e non vi sono dati nuovi alla porta di I/O 0; se il risultato non è zero, il bit 0 era a 1, cioè vi sono dati nuovi alla porta di I/O 0.

Ricordate che lo stato Z registrerà se l'istruzione NIA genera un risultato uguale a zero o no.

Dopo la lettura dei dati dalla porta di I/O 0, il programma può resettare il bit 0 della porta di I/O 2 a 0, e può settare il bit 1 a 1, usando queste quattro istruzioni:

INS	2	Prende in input lo stato
NIA	H'FE'	Azzera il bit 0
OIA	H'02'	Posiziona il bit 1 a 1
OUTS	2	Mette in output il risultato

Ecco che cosa succede:

		Contenuto di A0	Contenuto della Porta di I/O 2
INS	2	XXXXXXXX	XXXXXXXX
NIA	H'FE'	XXXXXXX0	XXXXXXXX
OIA	H'02'	XXXXXX10	XXXXXXXX
OUTS	2	XXXXXX10	XXXXXX10

X rappresenta di nuovo una qualsiasi cifra binaria (0 o 1).

Se non vi è chiaro come funzionano AND e OR, ritornate al Capitolo 2. Tutto quello che facciamo è eseguire un'operazione di AND sul contenuto della porta di I/O 2 con 11111110, poi un'operazione di OR del risultato con 00000010.

ISTRUZIONI DI SALTO CONDIZIONATO

Fino ad ora, i flag di stato Zero (Z), Carry (C), Overflow (O) e Segno (S) sono stati delle inutili curiosità, perchè il microcomputer non forniva alcun modo di trarne vantaggio.

Qual'è il modo di usare i flag di stato?

La risposta è: fornire istruzioni che permettano alla sequenza di esecuzione del programma di dipendere dalla condizione di un flag di stato.

Abbiamo già visto due esempi di come i flag di stato possono determinare il corso successivo dell'esecuzione del programma. Nella descrizione delle istruzioni operative in modo immediato, al bit 0, se la porta di I/O è 2, si controlla per vedere se c'è un valore zero o non zero. Se questo bit ha un valore di zero, l'esecuzione del programma deve saltare a qualche sequenza di istruzioni che non tenti di leggere dei nuovi dati dalla porta di I/O 0. Se questo bit è 1, la sequenza di esecuzione del programma deve saltare ad una routine che prenda in input i dati dalla porta di I/O 0.

La discussione sulle categorie delle istruzioni Load e Store è iniziata con una routine che carica la lunghezza del buffer nell'accumulatore A1, poi decrementa il contenuto di A1 come mezzo per provare se l'ultimo byte del buffer è stato spostato; fino a che A1 non è decrementato a zero, l'esecuzione del programma ritorna all'inizio del loop di spostamento dei dati; non appena il contenuto di A1 viene decrementato a zero, l'esecuzione deve continuare e non saltare indietro:

```
        LIM   A0,LENGTH  Carica la lunghezza del buffer
        LIM   DC0,BUFA   Carica l'indirizzo di inizio del buffer sorgente
        LIM   DC1,BUFA   Carica l'indirizzo di inizio del buffer destina-
                         zione
LOOP    LNA   DC0        Carica il byte dati in input seguente, incre-
                         mentare DC0
        SNA   DC1        Memorizzare il byte dati in input seguente,
                         incrementare DC1
        AIB   H'FF'      Aggiungere H'FF' ad A1; questo decremen-
                         ta A1
        Se A1 non contiene 0, ritorna a LOOP
        Se A1 contiene 0, continua con l'istruzione successiva.
```

Mentre l'istruzione AIB (che abbiamo già descritto) in effetti decrementa il contenuto di A1, un'istruzione operativa su un registro (che non è ancora stata descritta) viene codificata con un byte, anzichè con due.

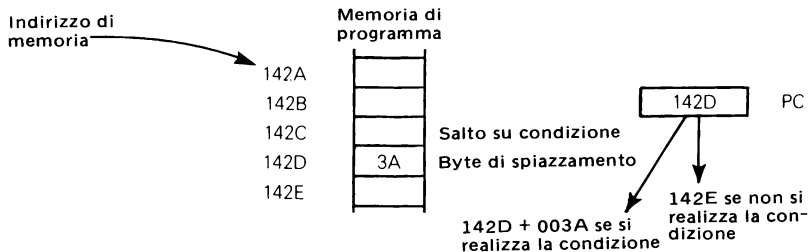
GIUSTIFICAZIONE DELL'ISTRUZIONE DI SALTO CONDIZIONATO

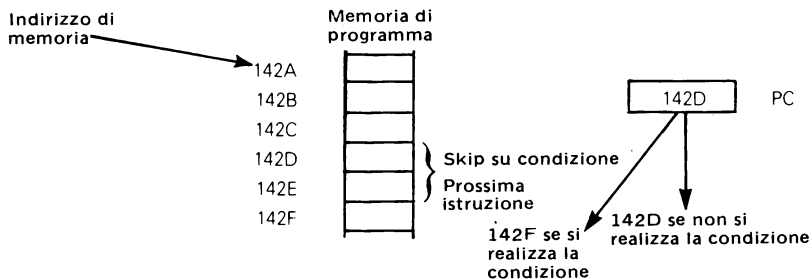
Le istruzioni di salto condizionato sono vitali per un microcomputer perchè esse sono il mezzo per controllare i flag di stato; i flag di stato, a loro volta, sono vitali per un microcomputer perchè sono il mezzo per determinare che cosa succede quando viene eseguita un'istruzione con più di un possibile risultato.

Vi sono due filosofie rispetto le istruzioni di salto condizionato; una usa i salti, l'altra usa gli skip.

FILOSOFIE DEL SALTO

Un'istruzione di salto condizionato, come implica il nome, ha uno spiazzamento di uno o due byte che segue il codice istruzione (proprio come i dati immediati). Se si verifica una certa condizione lo spiazzamento viene sommato al contenuto del Program Counter come numero binario con segno, e viene così eseguito un salto di programma. Se la condizione specifica non si verifica, il Program Counter viene incrementato normalmente e viene eseguita l'istruzione che segue in sequenza.

Ciò può essere illustrato in questo modo:

FILOSOFIA DELLO SKIP

L'istruzione di skip condizionato è seguita dallo spiazzamento. La logica di questa istruzione stabilisce che, se si verificano le condizioni di stato specificate, verrà effettuato uno skip della istruzione sequenziale successiva; se le condizioni specificate non si verificano, verrà eseguita l'istruzione sequenziale successiva. Ecco come lo si può illustrare:

Nell'illustrazione suddetta, osservate che se l'istruzione seguente fosse un'istruzione di due byte, il Program Counter dovrebbe essere incrementato a $142E_{16}$, se la condizione si verificasse. Se "l'istruzione seguente" fosse un'istruzione di tre byte, il Program Counter dovrebbe essere incrementato a 1430_{16} se la condizione si verificasse.

Inserendo un'istruzione di salto incondizionato direttamente dopo uno skip sull'istruzione, si viene ad avere il contrario di un'istruzione di salto condizionato:

Osservate che la logica di autoincremento e dello skip disponibile con le istruzioni di Load e di Store è una forma di istruzione di skip condizionato.

Il nostro microcomputer dovrebbe comprendere istruzioni di salto condizionato o di skip condizionato? Sceglieremo le istruzioni di salto condizionato perchè sono un po' più economiche con questi due tipi di sequenza di esecuzione:

```
        Ciclo di programma spostamento dati
LOOP    LNA     DC0                 LOOP    LNA     DC0
        SNA     DC1                         SNA     DC1
        AIB     H'FF'                       AIB     H'FF'
        Salto a ciclo se A1 = 0             Skip dell'istruzione successiva
                                            se A1 non è uguale a 0
                                            JMP     LOOP
        Logica di salto                     Logica di skip
```

SALTO A QUALI CONDIZIONI?

Quali sono le condizioni alle quali effettueremo il salto? Sceglieremo le otto condizioni di salto seguenti:

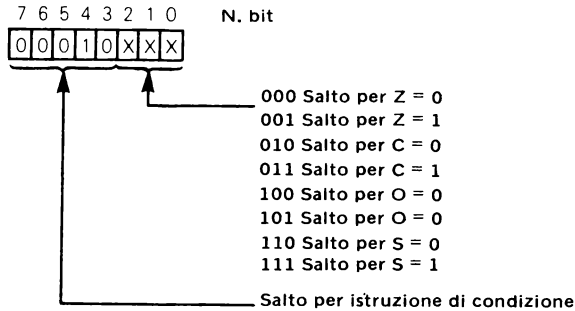
Salto per Zero (Z) uguale a 0
Salto per Zero (Z) uguale a 1
Salto per Carry (C) uguale a 0
Salto per Carry (C) uguale a 1
Salto per Overflow (O) uguale a 0
Salto per Overflow (O) uguale a 1
Salto per Segno (S) uguale a 0
Salto per Segno (S) uguale a 1

Le istruzioni di salto condizionato saranno seguite da uno spiazzamento di un solo byte, il che significa che è possibile uno spiazzamento in avanti o all'indietro di + 127 o - 128 byte. Ciò è ragionevole, dal momento che il 90% o più di tutti i salti vengono modificati da questo range, perciò, fornendo uno spiazzamento di due byte si sprecherebbe memoria. Naturalmente, potete sempre generare un salto di range più lungo combinando un'istruzione di salto incondizionato con una di salto condizionato, così:

```
        Salto per Z = 0
        Spiazzamento a THERE. Fuori dal range!
Sostituire:
        Salto a HERE per Z = 1
        JMP     THERE
HERE Istruzione successiva
```

L'istruzione JMP è seguita da un indirizzo di due byte, perciò essa può continuare ad essere eseguita in qualunque punto della memoria.

La sequenza di istruzioni di salto illustrata in precedenza ha la stessa logica di una istruzione di skip condizionato.

Useremo gli otto seguenti codici oggetto per le nostre otto istruzioni di salto condizionato:

Le istruzioni di salto condizionato avranno il formato:

OP LABEL

LABEL è l'etichetta dell'istruzione che deve essere eseguita se si verifica la condizione specificata da OP.
L'Assembler potrebbe convertire LABEL in uno spiazzamento sottraendo il contenuto del Program Counter dal valore dell'indirizzo di 16 bit assegnato a LABEL: se il risultato è fuori dal range, l'Assembler stamperà un messaggio di errore.

OP sarà uno dei campi codice menmonico seguente:

BZ	Salto se Zero (Z = 1)
BNZ	Salto se Non Zero (Z = 0)
BC	Salto se Carry (C = 1)
BNC	Salto se Non Carry (C = 0)
BO	Salto se Overflow (O = 1)
BNO	Salto se Non Overflow (O = 0)
BP	Salto se Positivo (S = 0)
BN	Salto se Negativo (S = 1)

SALTO PER MINORE, UGUALE O MAGGIORE

Le istruzioni di confronto manderanno parecchio in confusione i programmatori principianti. "Salto per zero" e "salto per Carry" non hanno significato diverso "salto per maggiore" o "salto per minore".
Ricordate che l'istruzione di confronto sottrae un operando dal contenuto dell'accumulatore specificato, e posiziona i flag di stato basandosi sul risultato della sottrazione. Possono perciò verificarsi le seguenti condizioni identificate:

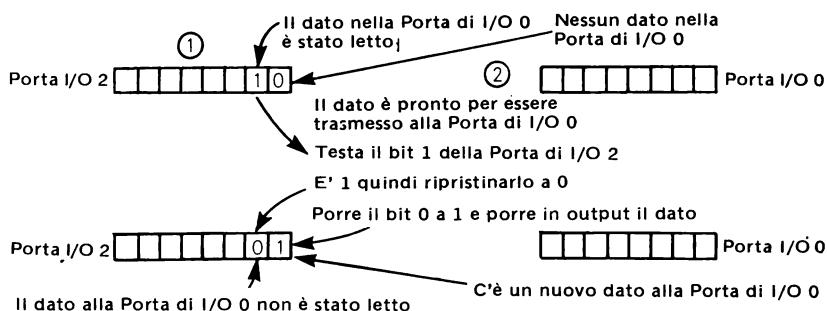
Salto per accumulatore minore o uguale (BLE)
Salto per accumulatore minore dell'operando (BL)
Salto per accumulatore minore e operando uguali (BE)
Salto per accumulatore minore e operando non uguali (BNE)
Salto per accumulatore minore maggiore dell'operando (BG)
Salto per accumulatore minore maggiore o uguale (BGE)

A seconda se il contenuto dell'accumulatore viene interpretato come dati binario con segno o senza segno la logica dei salti condizionati può essere determinata usando la seguente logica Booleana:

Condizione di salto	Condizione booleana	
	Dati con segno	Dati senza segno
BLE	Z OR (S XOR O) = 1	C = 0 OR Z = 1
BL	S XOR O = 1	C = 0
BE	Z = 1	Z = 1
BNE	Z = 0	Z = 0
BG	Z OR (S XOR O) = 0	C = 1 OR Z = 0
BGE	S XOR O = 0	C = 1

Nella tabella sopra descritta, per i dati senza segno, alcuni microcomputer invertono lo stato di Carry in seguito ad un'operazione di sottrazione o confronto. In questo caso bisogna cambiare C=1 e C=0.

Allo scopo di illustrare l'uso delle istruzioni di salto condizionato, daremo un'altra occhiata al modo con cui il controllore della temperatura della doccia potrebbe leggere i dati che vengono messi in input dal termometro.

Quando il termometro è pronto per mettere in output un byte dati, controlla il bit 1 della porta di I/O 2. Se questo bit è 1, la logica del termometro suppone che i dati che esso ha mandato in precedenza siano stati letti ed elaborati; perciò, la logica del termometro trasmette un nuovo bit dati alla porta di I/O e segnala questo evento posizionando il bit 0 della porta di I/O 2 a 1. La logica del termometro resetterà anche il bit 1 della porta di I/O 2 a 0, perchè i dati alla porta di I/O 0 non sono stati ancora letti:

Per leggere i dati messi in input dal termometro, il programma del microcomputer deve continuare a controllare i bit della porta di I/O 2 fino a che questo bit viene sentito a 1. Poi il microcomputer deve leggere i dati della porta di I/O 0, ma allo stesso tempo deve resettare il bit 0 della porta di I/O 2 a 0, dato che, non appena i dati vengono letti nella porta di I/O 0, esso diventa un dato vecchio. Il programma deve ora settare il bit 1 della porta di I/O 2 a 1; questo dice alla logica del termometro che i dati nella porta di I/O 0 sono stati letti.

La seguente sequenza di istruzioni esegue le operazioni ora descritte; inoltre, essa suppone che il byte di dati letto alla porta di I/O 0 venga memorizzato in una posizione di memoria indirizzata dal Data Counter DC0. Con DC0 si usa l'indirizzamento di

autoincremento, cosicchè il Data Counter indirizza automaticamente il byte seguente del buffer dati in input, pronto per l'accesso successivo alla porta di I/O 0.

```
LOOP    INS     2       Prende in input lo stato
        NIA     H'01'   Azzera tutti i bit tranne il bit 0
        BZ      LOOP    Ritorna a LOOP se il bit 0 è 0
        INS     2       I nuovi dati sono pronti. Prende ancora in input
                        lo stato
        NIA     H'FE'   Resetta il bit 0 a 0
        OIA     H'02'   Setta il bit 1 a 1
        OUTS    2       Rimemorizza il nuovo stato alla porta di I/O 2
        INS     0       Prende in input il byte dati
        SNA     DC0     Memorizza usando l'indirizzamento implicito,
                        di autoincremento.
```

STATO CONDIZIONATO AD UNA SUBROUTINE

Alcuni microcomputer hanno delle istruzioni di salto ad una subroutine simili alle istruzioni di salto condizionato che abbiamo appena descritto. Il nostro microcomputer ha un'istruzione di salto ad una subroutine che è stata descritta come un'istruzione di tipo immediato.

Le istruzioni di salto condizionato ad una subroutine saranno di solito seguite da un indirizzo di due byte, dato che le subroutine possono anche risiedere in memorie molto lontane dalle istruzioni di salto. La logica delle istruzioni di salto condizionato ad una subroutine è per il resto simile al salto condizionato: se si verifica la condizione specificata, avviene il salto alla subroutine; se no, viene eseguita l'istruzione successiva.

RIENTRO CONDIZIONATO DA UNA SUBROUTINE

Molti minicomputer hanno anche un set di istruzioni di rientro condizionato da una subroutine. Queste istruzioni rimemorizzano nei Program Counter l'indirizzo conservato dall'istruzione di salto alla subroutine. Non abbiamo nessuna speciale istruzione di rientro dalla subroutine, useremo al suo posto un'istruzione Pop (descritta insieme alle istruzioni dello stack).

ISTRUZIONI DI SPOSTAMENTO DA REGISTRO A REGISTRO

Vi sono due tipi di istruzioni che fanno riferimento a due registri della CPU; le istruzioni che spostano i dati da un registro all'altro, e le istruzioni che eseguono operazioni di tipo secondario di riferimento alla memoria, ma completamente all'interno della CPU.

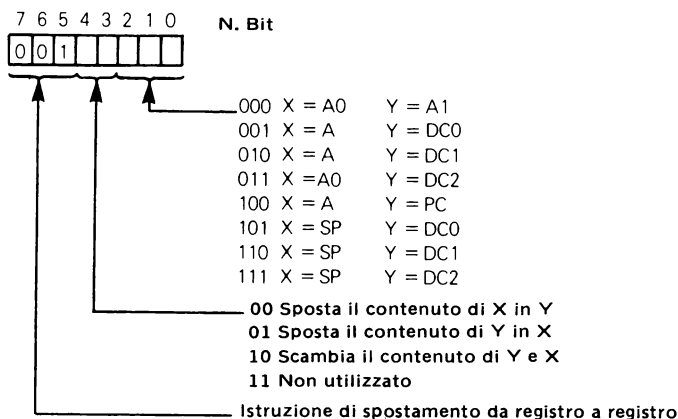
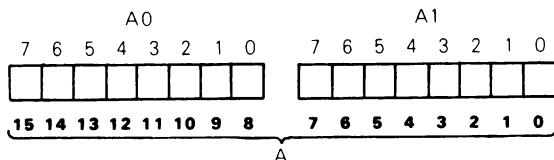
GIUSTIFICAZIONE DELLE ISTRUZIONI DI SPOSTAMENTO DA REGISTRO A REGISTRO

Le istruzioni che spostano i dati da registro a registro possono essere molto limitate nel nostro microcomputer, data la sua organizzazione dei registri. Dobbiamo essere in grado di spostare i dati fra A0 ed A1. Anche lo scambio del contenuto degli accumulatori è spesso utile.

Lo spostamento dei dati dagli accumulatori ai Data Counter permette alla logica di programma di creare indirizzi variabili negli accumulatori e poi di spostarli in un Data Counter per l'indirizzamento implicito variabile. Lo spostamento dei dati nella direzione inversa permette ad un Data Counter di essere usato come memoria temporanea dei dati degli accumulatori; naturalmente, si suppone con ciò che il Data Counter in questione non venga usato per l'indirizzamento implicito.

STACK MULTIPLI Raramente c'è bisogno di spostare i dati da un Data Counter ad un altro. Comunque, la possibilità di spostare i dati fra lo Stack Pointer e i Data Counter, o fra lo Stack Pointer e gli accumulatori è utile, dato che questo permette ad un programma di avere più di uno stack. DC2 potrebbe essere usato come buffer per lo Stack Pointer; ad esempio, ora, scambiando i contenuti di DC2 e di SP, si potrebbe accedere a due stack.

SALTO CALCOLATO Lo spostamento dei dati fra gli accumulatori e il Program Counter permette alla logica di programma di calcolare gli indirizzi di salto. Ciò è molto utile nelle tabelle di salto, che vengono illustrate più avanti in questo capitolo.

Forniremo quindi istruzioni di spostamento dati e istruzioni di scambio dati, come segue:

Nella suddetta descrizione, X = A specifica il valore di 16 bit formato dai due accumulatori, così:

SPOSTAMENTO **Questa istruzione:**

MOV S,D

sposterà il contenuto del registro specificato da S nel registro specificato da D. S e D devono essere una delle otto coppie valide che abbiamo illustrato; perciò questi spostamenti sono legali:

MOV	A1, A0	Spostare il contenuto di A1 in A0
MOV	A0, A1	Spostare il contenuto di A0 in A1
MOV	SP, DC1	Spostare il contenuto dello Stack Pointer in DC1

Questo spostamento è illegale:

MOV DC1, DC0

L'operazione desiderata può essere eseguita attraverso questi due spostamenti legali:

MOV	DC1,A	Sposta il contenuto di DC1 negli accumulatori
MOV	A; DC0	Sposta gli accumulatori in DC0

Ricordate il programma di controllo del cambiamento degli switch; usava un'istruzione di spostamento da registro a registro, così:

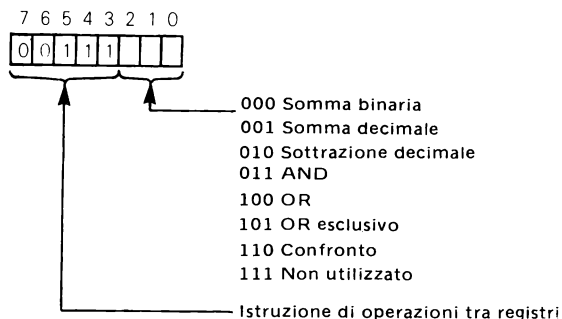
IN	4	Prende in input le nuove posizioni degli switch
XRA	SWITCH	Identifica gli switch cambiati
MOV	A0, A1	Salva il contenuto di A0 in A1
ANA	SWITCH	Identifica gli switch che sono stati attivati

SCAMBIO **I campi codice mnemonico delle istruzioni di scambio saranno:**

X S,D

Valgono per S e D le stesse regole descritte per MOV.

ISTRUZIONI DI OPERAZIONI TRA REGISTRI

Dato che il nostro microcomputer ha un numero di istruzioni di riferimento secondario alla memoria, ha bisogno di pochissime istruzioni di operazione tra registri; basteranno le sette istruzioni seguenti, che sono parallele alle istruzioni di riferimento secondario alla memoria:

I quattro flag di stato vengono settati o risettati per riflettere i risultati dell'operazione.

Dato che c'è un'istruzione di scambio A0-A1, abbiamo solo un set di istruzioni di operazione da registro a registro, dove A0 è sempre la destinazione del risultato.

ADDIZIONE BINARIA
ADDIZIONE DECIMALE
SOTTRAZIONE DECIMALE
AND
OR
OR ESCLUSIVO
CONFRONTO

Le istruzioni di operazione tra registri useranno questi campi codice mnemonico:

AB	Somma binaria di A1 con A0
AD	Somma decimale di A1 con A0
SD	Sottrazione decimale di A1 da A0
AND	AND di A1 con A0
OR	OR di A1 con A0
XOR	XOR di A1 con A0
CMP	Confronto tra A1 e A0

Nessuna delle istruzioni di operazione tra registri ha gli operandi.

Queste tre istruzioni permetteranno ad A1 di essere la destinazione di qualunque istruzione di operazione tra registri:

X	A0, A1	Scambia il contenuto di A0 e A1
AB		Addizione binaria; il risultato è in A0
X	A0, A1	Scambia il contenuto di A0 e A1

GIUSTIFICAZIONE DELLE ISTRUZIONI DI OPERAZIONE TRA REGISTRI

Tali istruzioni sono convenienti, ma non vitali, dato che esse non fanno niente che possa essere fatto usando le istruzioni di Load, di Store e le istruzioni secondarie di riferimento alla memoria. Esse saranno eseguite più velocemente delle istruzioni di riferimento secondario alla memoria, dato che queste ultime richiedono che un byte dati venga prelevato dalla memoria — e questo porta via del tempo.

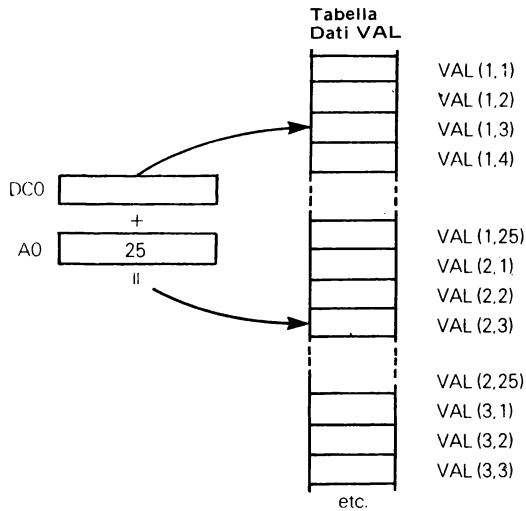
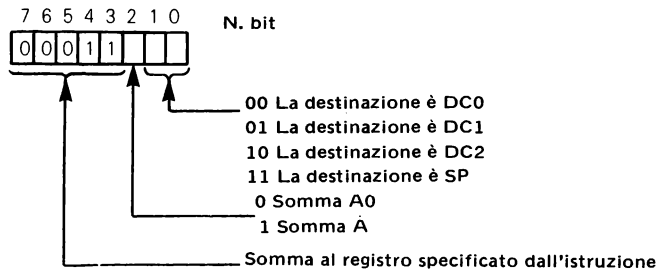
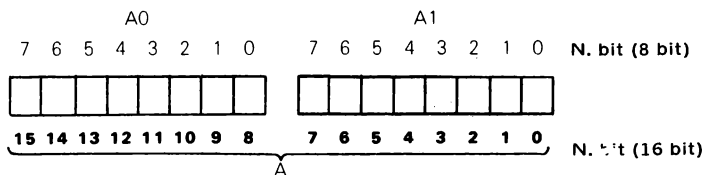
ADDIZIONE DELL'ACCUMULATORE AL DATA COUNTER

Vi è un altro set di istruzioni di operazioni tra registri che si dimostrerà molto utile; permetteremo al contenuto dell'accumulatore A0 di essere addizionato come numero binario con segno, ad uno qualunque dei Data Counter. Ciò permette di calcolare uno spiazzamento dell'indirizzo dei dati, e poi di sommarlo (o sottrarlo) ad un Data Counter.

MATRICI D'INDIRIZZAMENTO

Questa istruzione è particolarmente utile nell'aritmetica matriciale, dove si possono usare i parametri con doppi indici, quali:

VAL (X,Y)

Se la dimensione di Y è nota, si può trattare ogni incremento di X aggiungendo la dimensione di Y al Data Counter che sta indirizzando VAL. Vediamo nella figura di pagina seguente come si può illustrare.

Per estendere questo tipo di trattamento della matrice, **permetteremo anche che A0 ed A1, vengano trattati come un'unità di 16 bit, e vengano sommati ad un qualsiasi Data Counter. Abbiamo ora questi codici istruzione:**

A specifica un'unità di 16 bit:

Useremo questi campi codice mnemonico:

DAD S,D

S è la sorgente, e può essere A0 o A; non sono permesse altre scelte.

D è la destinazione, e può essere DC0, DC1, DC2 o SP; non sono permesse altre scelte.

TABELLE DI SALTO

Anche l'istruzione di addizione accumulatore-Data Counter è utile per creare tabelle di salto. Una tabella di salto è un elenco di indirizzi che identificano un numero di programmi, di cui uno solo deve essere eseguito, in base alla logica del programma in corso.

Prima creeremo una tabella degli indirizzi iniziali dei programmi: dato che questo non è un concetto semplice, lo illustreremo con un esempio che usa dei numeri reali, ma arbitrari:

```
ADDR1   EQU    H'1247    Inizio del programma 1
ADDR2   EQU    H'183C'   Inizio del programma 2
ADDR3   EQU    H'28CA'   Inizio del programma 3
        etc.
        ORG    H'0800'
BTBL    DA     ADDR1
        DA     ADDR2
        DA     ADDR3
        etc.
```

DIRETTIVE DI ASSEGNAMENTO

DI DEFINIZIONE DELL'INDIRIZZO

DI ORIGINE

Ricordate che il campo codice mnemonico EQU è una direttiva di Assembler;esso dice all'Assembler quali valori assegnare ai simboli ADDR1, ADDR2, ADDR3, ecc.

Il campo codice mnemonico DA è una direttiva di Assembler di "definizione d'indirizzo"; esso dice allo Assembler di mettere il valore di 16 bit fornito dallo operando nelle due posizioni di memoria seguenti.

Il campo codice mnemonico ORG è una direttiva di Assembler che fornisce l'indirizzo di memoria corrente. In questo caso, esso definisce l'indirizzo di memoria attuale 0800_{16}; in memoria, le istruzioni suddette risultano come in questi sei byte di dati:

Da notare che l'etichetta BTBL diventa un simbolo con il valore 0800_{16}.
Ora supponiamo che un numero di programma si trovi nell'accumulatore A0; possiamo eseguire il programma identificato dal numero di programma in questo modo:

```
LIM   DC0, BTBL   Carica l'indirizzo iniziale degli indirizzi di program-
                  ma in DC0
DAD   A0, DC0     Aggiunge due volte il numero degli elementi della
                  tabella
DAD   A0, DC0     Dato che ogni indirizzo occupa due byte
LNA   DC0         Carica l'indirizzo identificato da DC0
LMB   DC0
MOV   A, PC       Spostare questo indirizzo nel PC
```

Guardate che cosa succede:

1) L'istruzione LIM carica 0800_{16} in DC0
2) Supponiamo che l'accumulatore A0 contenga 2; le due istruzioni ADD aggiungono 4 a DC0, che ora contiene 0804_{16}.
3) L'istruzione LNA carica il contenuto della posizione di memoria 0804_{16} in A0, poi incrementa DC0. Ora A0 contiene 28_{16} e DC0 contiene 0805_{16}.
4) L'istruzione LMB carica il contenuto della posizione di memoria 0805_{16} in A1. Ora A1 contiene CA_{16}.
5) L'istruzione MOV sposta il valore $28CA_{16}$ nel Program Counter, forzando un salto nella posizione di memoria $28CA_{16}$.

Quando usereste una tabella di salto? Un esempio è dato nella descrizione delle istruzioni di interruzione.

ISTRUZIONI DI OPERAZIONE SU UN REGISTRO

Tali istruzioni modificano il contenuto di un solo registro; nessun altro registro viene modificato in nessun modo.

Alcune istruzioni di operazione su un registro sono assolutamente necessarie, mentre altre non sono niente di più che convenienti. Identificheremo quindi i modi in cui si può modificare il contenuto di un registro, e determinare se l'operazione è necessaria, o è solo una convenienza.

INCREMENTO E DECREMENTO

Il bisogno di incrementare e decrementare il contenuto dei registri è universale; in qualunque momento un registro contenga un contatore o un indice c'è la probabilità che dovrà essere incrementato o decrementato. Entro un certo limite, le possibilità di autoincremento e autodecremento dell'indirizzamento implicito rendono il bisogno di incrementare e decrementare i contatori dati meno vitale; è comunque una possibilità utile, dato che permette agli indirizzi di essere incrementati o decrementati selettivamente, cioè non sempre.

COMPLEMENTO

Dato che non abbiamo istruzioni di sottrazione binaria, è di importanza vitale che vi sia un'istruzione per complementare almeno uno degli accumulatori. Il fatto di complementare i Data Counter non ha alcuna utilità. Vedere il Capitolo 2 che tratta la.sottrazione col complemento a due.

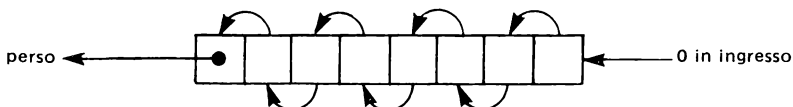
AZZERAMENTO DEI REGISTRI

Deve essere possibile azzerare tutti gli accumulatori; questo è un prerequisito fondamentale prima dell'esecuzione di un'addizione, o semplicemente un passo di inizializzazione. Non è necessario azzerare i registri indirizzi, dato che, come operazione più frequente, si usa il caricamento di dati che annulla il contenuto precedente.

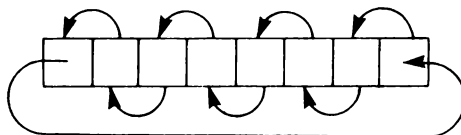
SCORRIMENTO SHIFT E ROTAZIONE

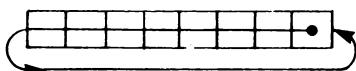
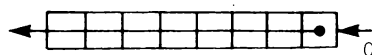
Le operazioni di scorrimento (Shift) e di rotazione sono molto importanti per due ragioni: sono vitali per la maggior parte degli algoritmi di moltiplicazione e divisione, e vengono usate spesso nelle operazioni di conteggio.

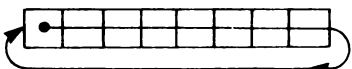
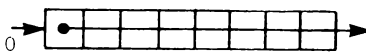
SHIFT

Un'operazione di scorrimento è un'operazione lineare:

Così un semplice scorrimento a sinistra, come mostra la figura, sposterà ogni bit nel bit successivo verso sinistra; il bit di ordine superiore, non avendo nessun bit alla sua sinistra, andrà perduto. Non essendoci bit alla destra del bit di ordine inferiore, verrà spostato uno 0 in quest'ultimo bit.

ROTAZIONE

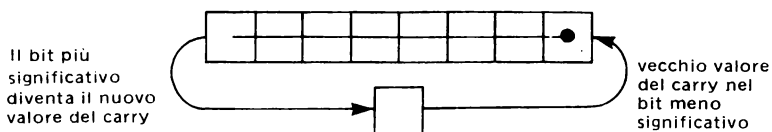
Un'operazione di rotazione è un'operazione circolare; si dovrebbe supporre che il bit di ordine inferiore e quello di ordine superiore sono adiacenti:

SHIFT SEMPLICE A ROTAZIONE

Sono possibili numerose varianti delle operazioni di scorrimento e di rotazione; potete far scorrere o ruotare a sinistra:

Rotazione semplice verso sinistra

Scorrimento shift semplice verso sinistra

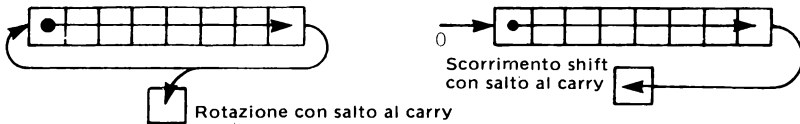
Potete far scorrere o ruotare a destra:

Rotazione semplice verso destra

Scorrimento shift semplice verso destra

Una rotazione (o uno scorrimento) a destra equivale alla divisione per 2, mentre la rotazione (o lo scorrimento) a sinistra equivale alla moltiplicazione per 2, e può essere riprodotta aggiungendo il contenuto di un registro a se stessa.

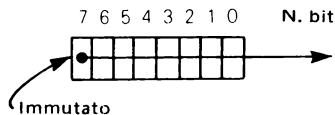
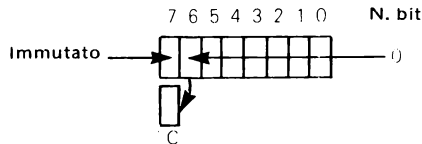
SCORRIMENTO E ROTAZIONE ATTRAVERSO IL CARRY

Scorrimento e rotazione possono avvenire attraverso lo stato Carry nel quale caso esse diventano operazioni identiche:

La figura illustra uno scorrimento o rotazione attraverso il Carry verso sinistra; l'equivalente verso destra si deduce automaticamente.

SCORRIMENTO E ROTAZIONE SALTO AL CARRY

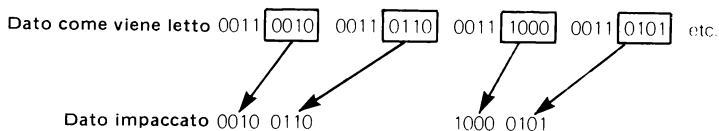
Secondo un'altra variante, un bit salta nello stato Carry, ma si esclude il vecchio stato Carry dallo scorrimento (o rotazione).

Questo scorrimento è molto utile come primo passo di scorrimento in un'operazione di scorrimento a più byte, dove si deve supporre un valore iniziale di 0 per lo stato Carry.

SCORRIMENTO ARITMETICO

Lo scorrimento può essere anche aritmetico e propagare il bit di ordine superiore (bit di segno) verso destra:

Uno scorrimento aritmetico verso sinistra manterrà il bit di segno e farà scorrere fuori il penultimo bit nel riporto:

LO SCORRIMENTO DI DATI IN BINARIO DECIMALE CODIFICATO

Uno scorrimento di quattro bit, a sinistra o a destra, è molto utile nelle applicazioni dei microcomputer che elaborano spesso dati numerici. Come è stato detto nel Capitolo 2, le cifre in binario decimale codificato occupano ognuna quattro bit; ogni byte contiene quindi due cifre BCD. Allora uno scorrimento di quattro bit equivale ad uno scorrimento di una sola cifra decimale, a sinistra o a destra; cioè, equivale a moltiplicare o dividere per dieci. Questo tipo di scorrimento facilita anche l'impaccamento e il disimpaccamento dei caratteri ASCII. Ricordate che la rappresentazione ASCII di una cifra decimale appare in questo modo:

0 = 00110000
1 = 00110001
2 = 00110010
3 = 00110011
4 = 00110100
5 = 00110101
6 = 00110110
7 = 00110111
8 = 00111000
9 = 00111001

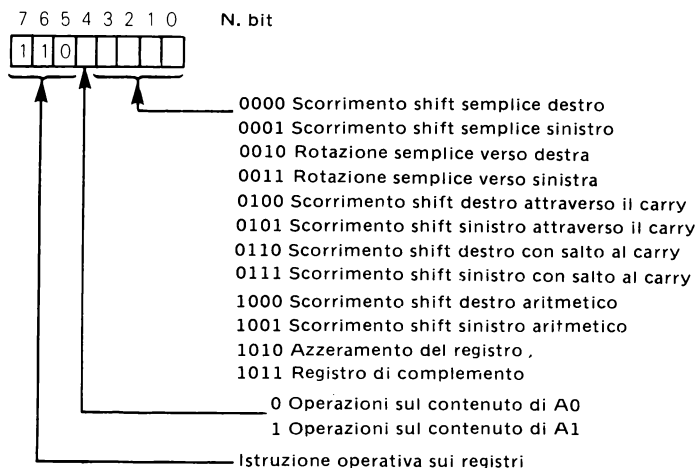
Supponiamo che una stringa di cifre ASCII venga letta attraverso una porta di I/O e debba essere impaccata in formato BCD, due cifre per ogni byte, così:

Lo scorrimento di 4 bit è naturale per questa operazione, stabiliremo dei campi codice mnemonico di scorrimento, poi scriveremo un programma per eseguire questa operazione che impacca i dati BCD.

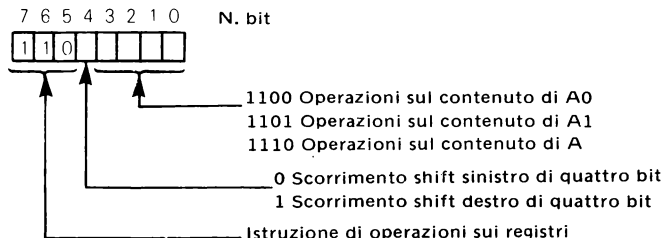
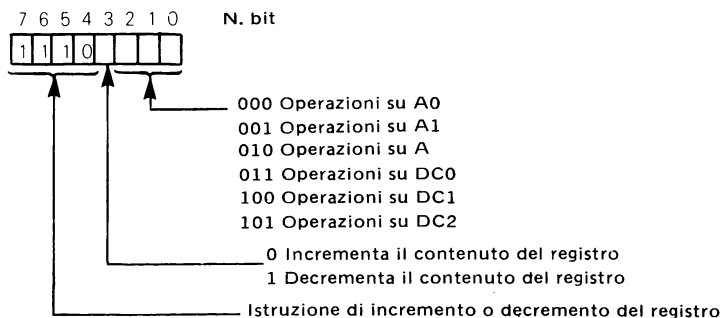
Quante e quali istruzioni di scorrimento/rotazione dovremmo avere?

Di solito, le istruzioni di scorrimento/rotazione sono rappresentate in modo inadeguato nei set di istruzioni dei microcomputer. Avremo tali istruzioni solo per i due accumulatori, ma forniremo scorrimenti e rotazioni senza Carry (semplici), con Carry, e con salto al Carry. Lo scorrimento del contenuto del Data Counter non è molto utile; fornirebbe uno scorrimento di 16 bit, ma questo è un lusso cui dovremo rinunciare.

Includeremo due versioni dello scorrimento di quattro bit a sinistra e a destra. Una opererà sul contenuto dell'accumulatore A0 a A1; l'altra opererà sull'unità combinata come un numero di 16 bit. In entrambi i casi, dato che abbiamo a che fare con una unità di quattro bit, l'accumulatore sarà ignorato durante l'operazione di scorrimento.

Ora possiamo riassumere i codici oggetto dell'istruzione di operazione su un registro nel modo seguente.

Per le operazioni che sono limitate agli accumulatori, queste sono le istruzioni e i loro codici oggetto:

Le istruzioni di scorrimento e di rotazione possono modificare lo stato Carry. L'istruzione di complemento influenzerà lo stato Zero. Non verrà cambiato nessun altro flag di stato.

Le due istruzioni di scorrimento di 4 bit possono operare sulle unità a 16 bit A0, A1, A (A0-A1). I codici oggetto per queste istruzioni saranno:

Le istruzioni di incremento e decremento operano sugli accumulatori e sui registri indirizzo; useranno questi codici oggetto:

ISTRUZIONI DI SCORRIMENTO E ROTAZIONE

Le istruzioni di scorrimento e rotazione avranno questi campi codice mnemonico, senza operandi:

SHRA	Scorrimento semplice a destra del contenuto di A0
SHRB	Scorrimento semplice a destra del contenuto di A1
SHLA	Scorrimento semplice a sinistra del contenuto di A0
SHLB	Scorrimento semplice a sinistra del contenuto di A1
RORA	Rotazione semplice verso destra di A0
RORB	Rotazione semplice verso sinistra di A1
ROLA	Rotazione semplice verso sinistra di A0
ROLB	Rotazione semplice verso sinistra di A1
SRCA	Scorrimento con carry verso destra di A0
SRCB	Scorrimento con carry verso destra di A1
SLCA	Scorrimento con carry verso sinistra di A0
SLCB	Scorrimento con carry verso sinistra di A1
SRBA	Scorrimento verso destra con salto al carry di A0
SRBB	Scorrimento verso destra con salto al carry di A1
SLBA	Scorrimento verso sinistra con salto al carry di A0
SLBB	Scorrimento verso sinistra con salto al carry di A1
SRAA	Scorrimento aritmetico verso destra di A0
SRAB	Scorrimento aritmetico verso destra di A1
SLAA	Scorrimento aritmetico verso sinistra di A0
SLAB	Scorrimento aritmetico verso sinistra di A1

SR4A	Scorrimento verso destra di quattro bit di A0
SR4B	Scorrimento verso destra di quattro bit di A1
SL4A	Scorrimento verso sinistra di quattro bit di A0
SL4B	Scorrimento verso sinistra di quattro bit di A1
SR4	Scorrimento verso destra di quattro bit di A0 e A1
SL4	Scorrimento verso sinistra di quattro bit di A0 e A1

INCREMENTO DEL REGISTRO

Questi sono i campi codice mnemonico che useremo per le istruzioni di operazione su un registro:

INC R

Questo specifica l'istruzione "incrementare il registro"; R può essere A0, A1, A, DC0, DC1 o DC2.

DECREMENTO DEL REGISTRO

L'istruzione "decrementare il registro" differirà solo nel campo codice mnemonico, e precisamente:

DEC R

COMPLEMENTO AZZERAMENTO

Verranno applicati solo ai due accumulatori, ed avranno questi campi codice mnemonico:

CLA	Azzerare A0
CLB	Azzerare A1
COA	Complementare A0
COB	Complementare A1

Queste quattro istruzioni non hanno operando.

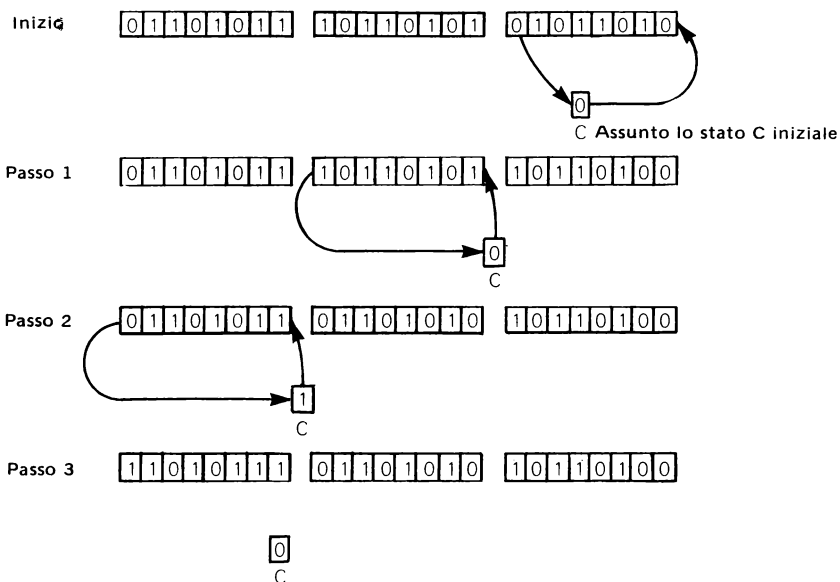
Illustreremo ora quanto valgono le istruzioni di operazione su un registro con alcuni esempi di come esse possono essere usate.

SCORRIMENTO A PIU' BYTE

Consideriamo gli scorrimenti a più byte; essi permettono che numeri a più byte vengano moltiplicati e divisi. Una rotazione con Carry propagherà uno scorrimento in un certo numero di byte, dato che il bit di ordine superiore di ogni byte si propaga nello stato Carry, poi nel bit di ordine inferiore del byte seguente. Possiamo illustrarlo con lo scorrimento semplice, verso sinistra, di tre byte.

Il programma per eseguire l'operazione è il seguente:

```
LIM    DC0, BUFA    Carica l'indirizzo iniziale del buffer in DC0
LMA    DC0          Carica il byte di ordine inferiore in A0 tramite DC0
SLBA                Fa scorrere verso sinistra con salto al carry
SDA    DC0          Rimemorizza il risultato; decrementa DC0
LMA    DC0          Carica il secondo byte in AC0
SLCA                Fa scorrere verso sinistra con carry
SDA    DC0          Rimemorizza il risultato; decrementa DC0
LMA    DC0          Carica l'ultimo byte
SLCA                Fa scorrere verso sinistra con carry
SMA    DC0          Rimemorizza il risultato
```

L'istruzione LIM carica semplicemente l'indirizzo dell'ultimo byte nel Data Counter DC0.

Le tre istruzioni seguenti, LMA, SLBA e SDA portano a termine il passo 1. Prima, LMA carica il byte di ordine inferiore nell'accumulatore A0 ma non modifica l'indirizzo in DC0, dato che vogliamo rimandare il risultato shiftato nello stesso indirizzo. L'istruzione SLBA è molto utile a questo punto, perchè esegue un salto al carry; non sappiamo che valore abbia lo stato Carry prima di entrare nella routine, ma non dobbiamo far caso all'istruzione SLBA; questa istruzione carica 0 nel bit di ordine inferiore di A0, e sposta il bit di ordine superiore di A0 nello stato Carry, pronto per essere fatto scorrere nel byte seguente. L'istruzione SDA sposta il contenuto shiftato di A0 di nuovo nel byte di memoria dal quale proveniene la sorgente non shiftata; poi viene decrementato l'indirizzo in DC0 per puntare al secondo byte.

Le tre istruzioni seguenti, LMA, SLCA e SDA eseguono il passo 2. Queste tre istruzioni differiscono dalle tre istruzioni precedenti solo per il fatto che ora deve essere eseguito uno scorrimento verso sinistra con Carry, dato che lo stato Carry rappresenta il bit di ordine superiore del byte precedente, che deve diventare il bit di ordine inferiore del byte attuale.

Il passo 3 è eseguito per mezzo delle ultime tre istruzioni, LMA, SLCA e SMA; queste tre istruzioni differiscono dalle tre istruzioni del passo 2 per il fatto che non dobbiamo preoccuparci di decrementare l'indirizzo in DC0, dato che non vi sono più byte da far scorrere.

Osservate che, dato che bisogna far scorrere solo tre byte, non usiamo un loop. Tutto il programma occupa solo 12 byte, tre per l'istruzione di caricamento immediato in DC0, uno per ognuna delle istruzioni rimanenti. Potremmo ridurre i tre passi in un set e se l'istruzione finale SMA diventa un'istruzione SDA. Il programma appare come segue.

Azzera lo stato Carry che deve essere inizialmente 0

```
        LIM   DC0,BUFA  Carica l'indirizzo iniziale del buffer in DC0
        LIM   A1,3      Carica il numero dei byte in A1
LOOP    LMA   DC0       Carica il byte seguente in A0, tramite DC0
        SLCA            Fa scorrere verso sinistra con Carry
        SDA   DC0       Rimemorizza il risultato, decrementa DC0
        DEC   A1        Decrementa il numero dei byte
        BNZ   LOOP      Rientra se non è la fine
```

Ora abbiamo un programma con otto istruzioni rispetto alle dieci di prima. Ma queste otto istruzioni occuperanno ancora 12 byte, tre per il caricamento immediato in DC0, due per ogni caricamento immediato in A1 e per il salto per non-zero, ed una per ogni istruzione rimanente. Questo è un altro esempio del fatto che, quando un loop ha pochissime iterazioni, una struttura di programma salto-e-ritorno non risulta molto economica se paragonata ad una struttura di programma a singola esecuzione.

CONTROLLO DEGLI SWITCH

Consideriamo ora i controlli degli switch. Gli otto switch che abbiamo descritto giustificando le istruzioni di riferimento secondario alla memoria, potrebbero essere controllati per lo stato di "on"o di "off" in un loop di programma, in questo modo:

1) Caricare 00000001 in A1. Useremo A1 come come contatore degli switch. Il suo contenuto verrà fatto scorrere verso sinistra con salto al Carry finchè appare un 1 nello stato Carry, cosa che indicherà che sono stati eseguiti otto scorrimenti.
2) Caricare la configurazione degli switch in A0.
3) Far scorrere A0 di un bit verso destra con salto al carry. Il bit di ordine inferiore di A0 è ora nello stato carry.
4) Conservare A0 ed A1 in DC2. Lo stato Carry riflette ancora il bit di ordine inferiore di A0, dato che un'istruzione di spostamento non influenzerà i flag di stato.
5) Salto per "Carry vero" al programma "switch in on". Altrimenti, continuare con il programma "switch off".
6) Quando il programma "switch on" o "switch off" ha completato l'esecuzione, ricaricare A0 ed A1 da DC2.
7) Far scorrere A1 verso sinistra di un bit con salto al Carry. Se il riporto è settato, è finito. Se il riporto non è settato, ritornare al passo 3.

I passi di programma richiesti per implementare la logica suddetta sono:

```
        LIM   A1,1      Carica 01 in A1
        IN    4         Prende in input la configurazione degli
                        switch dalla porta di I/O 4
LOOP    SRBA            Fa scorrere A0 verso destra con salto al
                        Carry
        MOV   A,DC2     Conserva A0 ed A1 in DC2
        BC    SWON      Salto per C = 1 al programma "switch on"
```

Questa è la logica del programma "switch off".

```
MOV   DC2,A   Rimemorizza A0 ed A1 da DC2
SLBB          Fa scorrere A1 verso sinistra con salto al Carry
```

IMPACCAMENTO DELLE CIFRE ASCII

Prendiamo ora in considerazione i passi di programma necessari per impaccare i bit numerici (i quattro bit di ordine inferiore) delle rappresentazioni di cifre numeriche ASCII; verranno impaccate due cifre numeriche in ogni

byte, come abbiamo descritto subito prima dell'illustrazione dei codici oggetto dello scorrimento.

Le cifre dovrebbero essere impaccate in questi passi:

Passo 1 — Leggere una cifra ASCII e memorizzarla nell'accumulatore A0.

Passo 2 — Far scorrere verso sinistra di quattro bit.

Passo 3 — Spostare il contenuto di A0 in A1, che contiene ora la cifra di ordine superiore, come segue:

Cifra ASCII: 0011XXXX
XXXX0000 Dopo lo scorrimento a sinistra di 4 bit.

Passo 4 — Mettere la cifra ASCII seguente nell'accumulatore A0.

Passo 5 — Mascherare i quattro bit di ordine superiore di A0, che contiene ora la cifra di ordine inferiore, come segue:

Cifra ASCII: 0011YYYY
0000YYYY Dopo aver mascherato i bit di ordine superiore.

Passo 6 — Aggiungere A1 ad A0, che contiene ora le cifre di ordine superiore e inferiore, come segue:

0000YYYY + XXXX0000 = XXXXYYYY

Passo 7 — Memorizzare le due cifre impaccate (supporremo che il buffer interessato sia indirizzato da DC1).

Passo 8 — Ritornare al passo 1 per le due cifre ASCII successive.

Supporremo che le cifre ASCII siano prese in input alla porta di I/O 5, e che il bit della porta di I/O 6 sia settato a 1 dal dispositivo di input quando ha trasmesso una cifra ASCII alla porta di I/O 5.

I passi di programma sono i seguenti:

```
LOOP1   IN      6        Prende in input lo stato
        SRBA             Fa scorrere il bit 0 di A0 con salto al Carry
        BNC     LOOP1    Se il riporto è 0, prende ancora in input lo
                         stato
        OUT     6        Se il riporto è 1, mettere in output A0 alla
                         porta di I/O 6
                         Questo azzera lo stato
        IN      5        Prende in input la cifra ASCII seguente
        SL4A             Fa scorrere a sinistra 4 bit
        MOV     A0, A1   Salva in A1
LOOP2   IN      6        Ripete le prime cinque istruzioni
        SRBA             Per la cifra ASCII seguente prendere in input
        BNC     LOOP2
        OUT     6
        IN      5
        NIA     H'0F'    Maschera i quattro bit di ordine superiore
        AB               Somma A1 ad A0
        LDA     DC0      Memorizza le due cifre impaccate
        JMP     LOOP1    Rientro per le due cifre successive
```

ISTRUZIONI SULLO STACK

Dato che il nostro microcomputer ha uno stack, esso deve avere delle istruzioni di Push per spostare il contenuto dei registri nello stack, deve avere anche delle istruzioni di Pop per spostare i dati dello stack nei registri.

SALTO AD UNA SUBROUTINE

Molti manuali di microcomputer mettono l'istruzione di salto ad una subroutine come un'istruzione dello stack, dato che essa spinge il contenuto del Program Counter nello stack prima di caricare un nuovo indirizzo nel Program Counter stesso.

PUSH

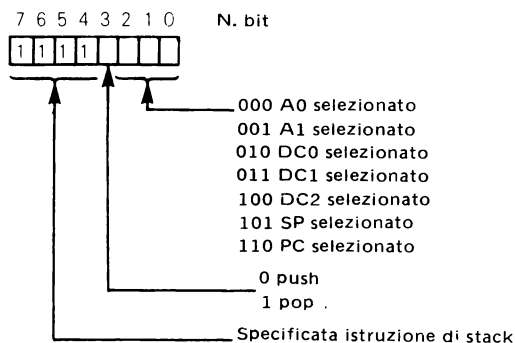
Le istruzioni di Push si useranno in primo luogo per l'elaborazione della interruzione; esempi di programma si trovano con le istruzioni del trattamento dell'interruzione.

POP – RIENTRO DALLE SUBROUTINE

Le istruzioni di Pop si usano nell'elaborazione delle interruzioni e per rientrare da una subroutine; vengono dati brevi esempi di questo secondo uso.

SUBROUTINE CON PASSAGGIO DI PARAMETRI

Le istruzioni di Push e di Pop vengono talvolta usate per passare dei parametri alle subroutine, illustreremo questo uso delle istruzioni di Push e di Pop più avanti.

Il nostro microcomputer avrà istruzioni di Push e di Pop che si riferiscono agli accumulatori e ai quattro registri indirizzo; i codici oggetto saranno i seguenti:

Le istruzioni di Push e di Pop useranno questo formato:

OP R

OP rappresenta il campo codice mnemonico dell'istruzione; esso sarà PUSH o POP, per un'istruzione di Push e di Pop, rispettivamente.

R specificherà il registro il cui contenuto deve essere messo nello stack, o che deve ricevere i dati estratti dallo stack. R può essere A0, A1, DC0, DC1, DC2 o PC. Non sono permessi altri simboli.

ISTRUZIONI DI RIENTRO

Potrà esserci un campo codice mnemonico in più per l'istruzione di rientro dalla subroutine. L'istruzione:

POP PC

sposterà i due byte della parte alta dello stack nel Program Counter, effettuando così un rientro dalla subroutine.

RET

Lo stesso codice oggetto; in altre parole, il campo mnemonico RET genererà il byte di un codice oggetto:

Per avere un esempio dell'uso delle istruzioni sullo stack, torniamo alla subroutine di spostamento dati che abbiamo descritto insieme alle istruzioni immediate; la subroutine era elencata così:

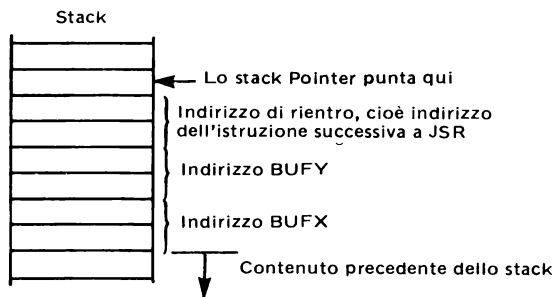
```
MOVE  LIM   DC0, BUFA   Carica l'indirizzo iniziale della sorgente
      LIM   DC1, BUFB   Carica l'indirizzo iniziale della destinazione
LOOP  LNA   DC0         Sposta i dati dalla sorgente
      SSA   DC1,.LOOP   alla destinazione
      Rientro dalla subroutine
```

PASSAGGIO DEI PARAMETRI

Dopo aver raggiunto l'istruzione di rientro, questa subroutine può diventare ancora più utile se gli indirizzi iniziali dei buffer sorgente e destinazione (BUFA e BUFB) sono variabili. Le istruzioni dello stack forniscono (ma non il migliore) un modo per farlo.

Prima di chiamare la subroutine MOVE, il programma può mettere il valore dei suoi indirizzi BUFA e BUFB nello stack, in questo modo:

```
LIM    DC0,BUFX
PUSH   DC0
LIM    DC0,BUFY
PUSH   DC0
JSR    MOVE
```

La parte dello stack appare ora così:

La subroutine MOVE deve essere modificata in questo modo:

```
MOVE  POP   DC2         Salva l'indirizzo di rientro in DC2
      POP   DC1         Carica BUFY come indirizzo iniziale della
                        destinazione
      POP   DC0         Carica BUFX come indirizzo iniziale della
                        sorgente
      PUSH  DC2         Sostituisce l'indirizzo di rientro nella parte
                        alta dello stack
LOOP  LNA   DC0         Sposta i dati dalla sorgente
      SSA   DC1,LOOP    alla destinazione
      RET               Estrae l'indirizzo di rientro dallo stack
```

ISTRUZIONI DI PASSAGGIO DEI PARAMETRI

Dato che le subroutine vengono usate così spesso, vale la pena di dare un'occhiata alle istruzioni che facilitano l'uso delle subroutine stesse.

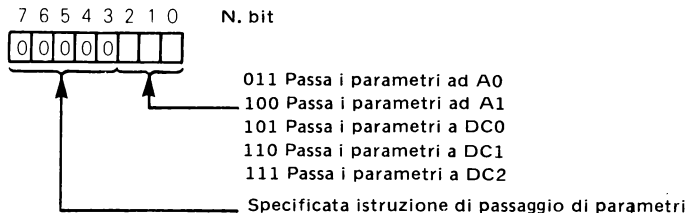
Torniamo di nuovo alla subroutine di spostamento dati che abbiamo sviluppato fino a questo punto.

In primo luogo, questa subroutine sposta semplicemente i dati da un buffer sorgente con un indirizzo predeterminato, ad un buffer destinazione con un altro indirizzo predeterminato.

PARAMETRI DELLE SUBROUTINE

In seguito, quando abbiamo descritto le istruzioni di Push e di Pop abbiamo migliorato la versatilità di questa subroutine permettendo al programma chiamante di specificare gli indirizzi del buffer sorgente e destinazione. Questi due indirizzi si chiamano "parametri", che il programma chiamante passa alla subroutine. Il passaggio dei parametri è una caratteristica molto importante del trattamento della subroutine; facilitando il passaggio dei parametri, il microcomputer diventa un dispositivo molto più efficace.
In effetti, le istruzioni di passaggio dei parametri sono molto semplici da specificare. Ciò che faremo è permettere ai parametri di venire dopo l'istruzione di salto alla subroutine, poi forniremo al microcomputer una forma di indirizzamento indiretto, in cui i due byte della parte alta dello stack diventano l'indirizzo di memoria dal quale verranno prelevati i dati.

Ma prima di spiegare questo concetto con figure ed esempi, definiamo le istruzioni di passaggio dei parametri che il nostro microcomputer comprenderà.

Prima di tutto, ci sono i codici oggetto che si devono usare:

Il formato dell'istruzione di passaggio dei parametri sarà il seguente:

SPP R

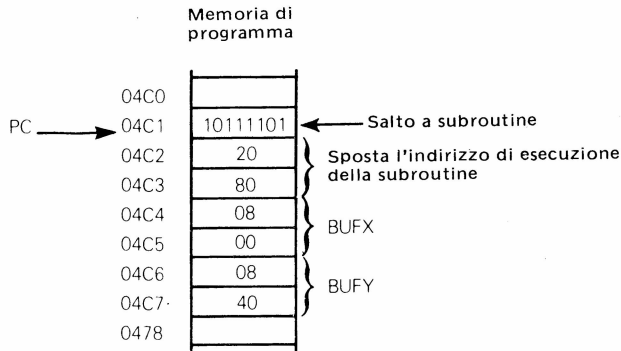
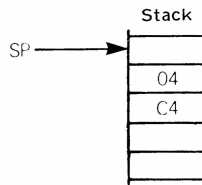
SPP è il campo codice mnemonico dell'istruzione; R identifica uno dei registri A0, A1, DC0 o DC2, non sono permessi altri simboli per R.

PASSAGGIO DEI PARAMETRI ALLE SUBROUTINE

Svilupperemo ora un'implementazione molto efficace della subroutine di spostamenti dati.

La subroutine verrà chiamata in questo modo:

```
JSR    MOVE     Chiama la subroutine di spostamento dati
DA     BUFX     Specifica l'indirizzo iniziale della sorgente
DA     BUFY     Specifica l'indirizzo iniziale della destinazione
```

Ricordate che il campo codice mnemonico DA rappresenta la direttiva di Assembler della definizione di indirizzo. Supponiamo che queste istruzioni risiedano in memoria nel modo che segue.

Dopo che è stata eseguita l'istruzione JSR, PC conterrà 2080_{16}, che è l'indirizzo di esecuzione della subroutine MOVE. Il precedente valore PC, $04C4_{16}$, sarà nella parte alta dello stack:

La subroutine MOVE appare in questo modo:

MOVE	SPP	DC0	Carica l'indirizzo iniziale della sorgente in DC0
	SPP	DC1	Carica l'indirizzo iniziale della destinazione in DC1
LOOP	LNA	DC0	Sposta i dati dalla sorgente
	SSA	DC1,LOOP	alla destinazione
	RET		Estrae l'indirizzo di rientro dallo stack

La prima istruzione SPP fa sì che la CPU esegua la seguente logica:

1) I due byte nella parte alta dello stack vengono prelevati e messi nella CPU.
2) Questi due byte sono trattati come un indirizzo di memoria. Il contenuto della posizione di memoria identificata da questo indirizzo è caricato nel byte di ordine superiore di DC0. Viene allora incrementato l'indirizzo di memoria. L'indirizzo di memoria era $04C4_{16}$, e la posizione di memoria $04C4_{16}$ contiene 08_{16}. Perciò, alla fine del passo il byte di ordine superiore di DC0 contiene il valore 08_{16} e l'indirizzo di memoria è stato incrementato a $04C5_{16}$.
3) Viene ripetuto il passo 2, con i dati prelevati dalla memoria che vanno al byte di ordine inferiore di DC0. Alla fine di questo passo, DC0 conteneva 0800_{16} e l'indirizzo di memoria è ora $04C6_{16}$.
4) L'esecuzione dell'istruzione è completa, cosicchè l'indirizzo di memoria viene rimandata nella parte alta dello stack, che contiene ora $04C6_{16}$, e non $04C4_{16}$.

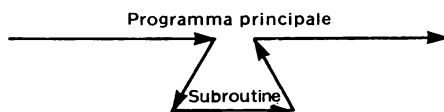
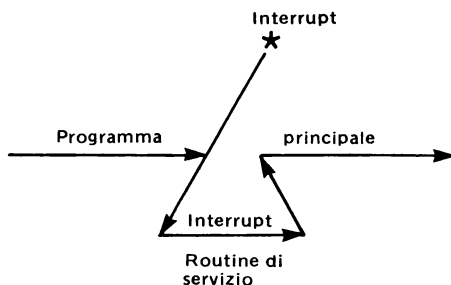
La seconda istruzione SPP è una ripetizione della prima istruzione SPP, ad eccezione del fatto che viene specificato come destinazione DC1, perciò, alla conclusione della

istruzione SPP, 0840_{16} sarà memorizzato in DC1, e i due byte della parte alta dello stack conterranno il valore $04C8_{16}$. Questo è l'indirizzo dell'istruzione che deve essere eseguita e che si trova dopo i due parametri, BUFX e BUFY. Alla conclusione della subroutine MOVE, l'istruzione RET estrarrà il valore $04C8_{16}$ e lo metterà nel Program Counter, permettendo così che continui la normale esecuzione del programma.

ISTRUZIONI DI INTERRUZIONE

In realtà parleremo di qualcosa in più, di queste istruzioni. Vi sono solo tre istruzioni d'interruzione; una disabilita tutte le interruzioni, la seconda abilita tutte le interruzioni, e la terza è un'istruzione di rientro dall'interruzione.

Come tratterà gli interrupt il nostro microcomputer?

Vi sono molte somiglianze fra l'elaborare un interrupt e l'entrare in una subroutine; in entrambi i casi, l'esecuzione del programma salta temporaneamente da un programma principale ad una sequenza logica secondaria, alla cui conclusione l'esecuzione ritorna al programma principale. La differenza fra una subroutine ed un'interruzione sta nel fatto che un salto ad una subroutine è parte della logica prestabilita per il programma:

Un interrupt, invece, è un evento non prestabilito e il programma principale non può sapere quando avverrà l'interruzione:

In un certo punto del Capitolo 5, abbiamo parlato dei vari modi in cui i dispositivi esterni possono interrompere la CPU. Ricordate che quando il protocollo di interruzione della CPU diventa più simile al minicomputer, e più sofisticato, così anche il costo e la complessità della logica esterna necessari per soddisfare le richieste del protocollo d'interruzione della CPU, aumentano. Adotteremo perciò uno schema molto semplice. I dispositivi di interruzione saranno messi col metodo Daisy-chain su di una sola linea di richiesta d'interruzione, e quando la CPU emette un segnale di riconoscimento, il dispositivo d'interruzione metterà in output un solo byte dati sulla porta di I/O con indirizzo FF_{16}. La CPU interpreterà i dati alla porta di I/O FF_{16} come se identificassero il dispositivo d'interruzione.

Non appena la CPU riconosce un interrupt, farà automaticamente tre cose:

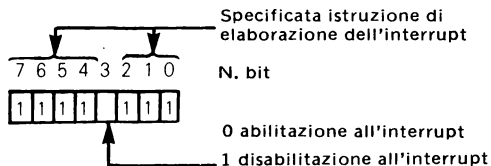
Primo: disabiliterà gli interrupt, impedendo così che un altro interrupt venga elaborato prima che sia adeguatamente trattato quello in corso. Un'istruzione di disabilitazione dell'interrupt deve essere eseguita dal programma prima che possa essere trat-

tata qualunque ulteriore interrupt.

Poi la CPU salverà il contenuto dei flag di stato mettendoli nello stack.

Infine la CPU metterà il contenuto del Program Counter sulla parte alta dello stack, ed azzererà il Program Counter. Ciò fa sì che l'esecuzione del programma continui dalla posizione di memoria 0.

Si può eseguire in qualunque momento un'istruzione di disabilitazione dell'interruzione per evitare che vengano riconosciute delle interruzioni; questa condizione durerà finchè l'istruzione di abilitazione all'interruzione non verrà eseguita.

Vediamo prima i codici oggetto delle istruzioni di abilitazione e disabilitazione dall'interrupt.

ABILITAZIONE DEGLI INTERRUPT

DISABILITAZIONE DEGLI INTERRUPT

I campi codice mnemonico per le due istruzioni sugli interrupt saranno:

DI

per la disabilitazione dell'interrupt, e

EI

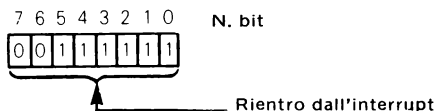
per l'abilitazione dell'interrupt.

RIENTRO DA UN INTERRUPT

L'istruzione di rientro dall'interrupt farà tre cose: prima di tutto ripristina i flag di stato, che erano stati salvati automaticamente nello stack quando era stato riconosciuto l'interrupt.

Poi estrarrà l'indirizzo di rientro dallo stack e lo metterà nel Program Counter.

Infine abiliterà gli interrupt.

Il codice oggetto di questa istruzione sarà:

Il campo mnemonico dell'istruzione sarà:

RTI

Per illustrare l'uso delle istruzioni sull'interrupt, mostreremo i passi di programma che seguono il riconoscimento dell'interrupt.

Mostreremo anche i passi di programma che devono essere messi alla fine di una routine di servizio dell'interrupt.

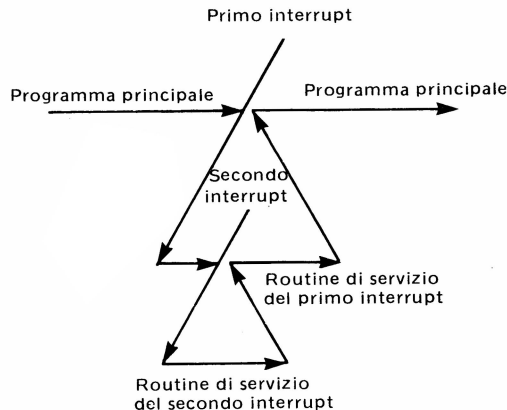
RICONOSCIMENTO DELL'INTERRUPT

Ecco che cosa deve accadere a seguito di un'interruzione:

1) Nel momento del riconoscimento dell'interrupt, la logica della CPU salva i flag di stato nella parte alta dello stack, mette il contenuto del Program Counter nella parte alta dello stack, poi disabilita gli interrupt. Il Program Counter è azzerato, il che significa che l'esecuzione del programma salta alla posizione di memoria 0.

2) A partire dalla posizione di memoria 0, c'è una breve sequenza di programma che salva il contenuto di tutti i registri della CPU mettendoli sullo stack. Ciò è necessario, dato che i registri possono essere usati in qualunque momento dal programma che sta per essere eseguito.

3) Dopo che tutti i contenuti dei registri sono stati messi da parte sullo stack, viene letto il contenuto della porta di I/O FF_{16}, ed usato per elaborare l'indirizzo di inizio del particolare programma che servirà il dispositivo d'interruzione identificato.

4) Il programma che viene eseguito seguendo il passo 3, volendo, può contenere una istruzione di abilitazione dell'interrupt. Se c'è questa istruzione, si può elaborare un'altro interrupt prima che il servizio all'interruzione in corso sia stato completato, in questo modo:

Se non sono abilitati gli interrupt non se ne possono elaborare altri finchè non viene eseguita l'istruzione di rientro dall'interrupt.

Questa è la sequenza dell'istruzione che, data la nostra logica di servizio dell'interruzione, deve essere presente a partire dalla posizione di memoria 0:

```
ORG    0
PUSH   A0          Salva i contenuti di tutti i registri
PUSH   A1          sullo stack
PUSH   DC0
PUSH   DC1
PUSH   DC2
IN     H'FF'       Prende in input identificativo di un
                   dispositivo della porta di I/O FF
LIM    DC0,BTBL    Carica l'indirizzo base della tabella di salto
SHLA               Fa scorrere semplicemente A0 a sinistra, per
                   moltiplicare x 2
DAD    A0,DC0      Somma A0 a DC0
LNA    DC0         Carica l'indirizzo di parte della routine di servizio
```

```
LMB    DC0     dell'interruzione
MOV    A,PC    Sposta l'indirizzo in PC
```

Ecco quello che fa il breve programma che avete visto.

SALVATAGGIO DEI REGISTRI SULLO STACK

Ricordate che il campo codice mnemonico ORG specifica l'indirizzo di memoria corrente per l'Assembler. Esso dice all'Assembler di incominciare a crèare il codice oggetto partendo dalla posizione di memoria 0.

Le cinque istruzioni di Push salvano i contenuti di tutti i registri nello stack.

L'istruzione IN riceverà l'identificativo (ID) di un dispositivo alla porta di I/O FF. Supponiamo che, nel tempo impiegato dalla CPU per eseguire l'istruzione di Push, il dispositivo d'interruzione sarà stato in grado di mettere il suo numero ID alla porta di I/O FF. Questo numero ID sarà nell'accumulatore A0.

TABELLA DI SALTO

Le istruzioni da LIM a MOV costituiscono una tabella di salto. Tali tabelle sono state descritte insieme all'istruzione di operazione tra registri DAD. Notate che, nelle sequenze delle istruzioni della tabella di salto precedente, è stata usata un'istruzione di scorrimento per moltiplicare il contenuto di A0 per 2 prima di aggiungerlo a DC0; nell'esempio di prima, l'istruzione DAD è stata eseguita due volte per ottenere lo stesso risultato.

L'indirizzo calcolato dalla tabella di salto diventa l'indirizzo iniziale della routine di servizio dell'interruzione, che verrà ora eseguita per servire il dispositivo specificato che ha richiesto l'interruzione. Una volta che la routine di servizio dell'interruzione ha completato l'esecuzione, chiamerà una subroutine che rovescerà i passi del riconoscimento d'interruzione, così: RINT, POP DC2 rimemorizza i contenuti di tutti i registri.

```
RINT   POP    DC2     Ripristino di tutti i contenuti dei registri
       POP    DC1
       POP    DC0
       POP    A1
       POP    A0
       RTI
```

IL RIPRISTINO DEI REGISTRI DALLO STACK

Osservate che i registri vengono estratti dallo stack nell'ordine contrario a quello con cui erano stati immessi, dato che lo stack è un'unità di memorizzazione "last-in-first-out".

L'istruzione finale RTI ripristina i flag di stato salvati (che è stato messo nello stack dopo il riconoscimento dell'interrupt), poi ricaricherà nel Program Counter l'indirizzo di memoria che era stato messo da parte durante il riconoscimento dell'interrupt.

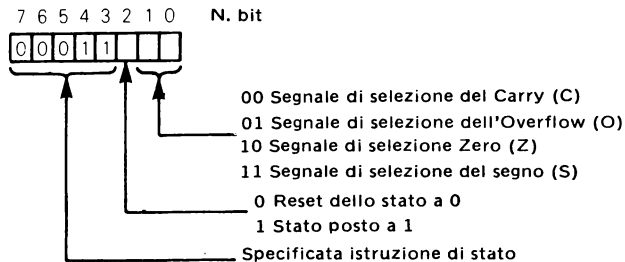
Se gli interrupt sono ancora disabilitati, l'istruzione RTI li riabiliterà.
L'esecuzione del programma continua ora dal punto in cui era stata interrotta.

ISTRUZIONI DI STATO

Dato che abbiamo quattro flag di stato, Segno, Carry, Overflow e Zero, deve essere possibile settare o risettare questi flag singolarmente. La situazione più comune in cui la logica di programma richiederà che un flag venga settato è appena prima di entrare in un loop di programma che contenga un'istruzione di salto condizionato all'inizio del loop stesso. Nel corso normale degli eventi, i flag di stato verranno settati più avanti nel loop per essere controllati quando la logica del programma torna indietro all'inizio del loop stesso. Deve essere possibile settare le condizioni di stato prima di

entrare nel loop, in modo che si possa arrivare al salto condizionato al primo passaggio senza problemi.

Vi sono anche molti algoritmi aritmetici a più byte che richiedono che gli stati Carry e Overflow vengano o azzerati o settati prima di iniziare l'algoritmo; in seguito, dopo che è stato elaborato ogni byte del numero a più byte, i riporti vengono fatti passare da un byte a quello seguente tramite due flag di stato, come descritto nel Capitolo 2.

Includeremo quindi queste otto istruzioni di stato:

L'istruzione per settare a 1 uno stato avrà questo formato:

SET X

L'istruzione per risettare a zero lo stato avrà questo formato:

RES X

In entrambi i casi, X può essere, C, O, Z o S, per identificare uno dei quattro flag di stato. Non sono permessi altri simboli.

Per fare un esempio dell'uso delle istruzioni sullo stato, ecco la routine di addizione binaria a più byte, descritta insieme alle istruzioni di riferimento secondario alla memoria, che inizia con l'azzeramento dello stato Carry:

```
        RES   C          Azzera lo stato Carry
LOOP    LMA   DC0        Carica il byte in input seguente
        ABA   DC1        Addizione binaria dal buffer del risultato
        SSA   DC1,LOOP   Memorizza il risultato, incrementa ed effet-
                         tua uno skip
```

Una volta nel loop, l'istruzione di addizione binaria ABA setta e risetta in modo appropriato lo stato Carry.

ISTRUZIONE DI ARRESTO

Ogni microcomputer ha un'istruzione di Alt. Quando essa viene eseguita, il microcomputer si ferma. In un minicomputer, o in un microcomputer con pannello frontale, l'esecuzione del programma viene ripresa premendo un apposito bottone sul pannello. Finchè si ha a che fare con la CPU, il segnale di reset che viene inserito nella CPU (che è stato descritto nel Capitolo 4) deve ricevere degli impulsi allo scopo di iniziare l'esecuzione dopo un'istruzione di arresto.

Nel nostro microcomputer, e in molti altri, il codice oggetto dell'istruzione di arresto consiste di tutti bit a 0. Questo viene fatto a ragion veduta, dato che le parole di me-

moria non usate contengono spesso tutti bit a 0. Nel caso che un programma, mentre viene messo a punto, faccia un salto errato e tenti di eseguire istruzioni in una certa area di memoria dove istruzioni non ne esistono, c'è una buona probabilità che trovi dei bit tutti a 0 e che cerchi di eseguirli come codice oggetto dell'istruzione seguente — e quindi il programma si fermerà e basta, e facendo meno danni possibile.

Il campo codice mnemonico dell'istruzione di arresto sarà, più o meno:

HALT

SOMMARIO DEL SET DI ISTRUZIONI

Vedrete che i libri che descrivono i singoli microcomputer forniscono tabelle che riassumono il set di istruzioni. Queste tabelle riassuntive sono molto utili. Supponendo di avere una conoscenza generale del linguaggio assemblatore, due o tre pagine vi dicono tutto quello che avete bisogno di sapere circa le operazioni eseguite quando viene trovata una qualunque istruzione.

Riassumeremo il nostro ipotetico set di istruzioni nella Tabella riassuntiva 7-1. Nel Volume 2 tabelle analoghe riassumeranno i set di istruzioni per microcomputer veri e propri.

Nella Tabella 7-1 vengono usati questi simboli:

AC0	Accumulatore AC0
AC1	Accumulatore AC1
ADDR	Un indirizzo di memoria di 16 bit
C	Stato Carry
DATA	Un'unità di dati binari di 8 bit
DC0	Data Counter DC0
DC1	Data Counter DC1
DC2	Data Counter DC2
DCX	Data Counter indefinito
DISP	Uno spiazzamento d'indirizzo binario con segno di 8 bit
DIST	Registro di destinazione indefinito
I	Indicatore di stato indefinito
O	Stato Overflow
P	Un numero di porta I/O
PC	Program Counter
R	Registro indefinito
S	Stato Segno
SP	Stack Pointer
SRC	Registro sorgente indefinito
SW	Stati
Z	Stato Zero
[ ]	Contenuto della posizione chiusa fra parentesi. Se fra parentesi vi è la designazione di un registro, si intende il contenuto del registro designato. Se fra parentesi c'è un numero di porta di I/O, allora si intende il contenuto della porta di I/O designata. Se fra parentesi c'è un indirizzo di memoria si intede il contenuto della posizione di memoria indirizzata.
[[ ]]	Indirizzo di memoria implicito; il contenuto della posizione di memoria designata dal contenuto di un registro.
∧	AND logico
∨	OR logico
∀	OR esclusivo
←	I dati sono trasferiti nella direzione della freccia
← →	I dati vengono scambiati fra le due posizioni rappresentate ai lati delle frecce

Sotto il titolo di STATI, nella Tabella 7-1, una X indica gli stati che vengono modificati nel corso dell'esecuzione delle istruzioni. Se non vi sono X, significa che lo stato mantiene il valore che aveva prima dell'esecuzione dell'istruzione.

Tabella 7-1. Sommario del set di istruzioni dell'ipotetico Microcomputer

TIPO	MNEMONICO	OPERANDO (I)	BYTE	STATI				OPERAZIONE EFFETTUATA
				C	O	Z	S	
I/O	INS,IN	P	1,2					[AC0]←[P] Input to A0 from I/O Port P (only 0, 1 or 2 for INS)
	OUTS,OUT	P	1,2					[P]←[AC0] Output from A0 to I/O Port P (only 0, 1 or 2 for OUTS)
Riferimento in memoria principale	LRA,LRB	ADDR	3					[AC0]←[ADDR], [AC1]←[ADDR] Load to A0 or A1, use direct addressing
	SRA,SRB	ADDR	3					[ADDR]←[AC0], [ADDR]←[AC1] Output from A0 or A1, use direct addressing
	LMA,LMB	DCX	1					[AC0]←[[DCX]], [AC1]←[[DCX]] Load AC0 or AC1 using implied addressing
	LNA,LNB	DCX	1					[AC0]←[[DCX]], [AC1]←[[DCX]], and [DCX]←[DCX] · 1 Load AC0 or AC1 using implied addressing with auto increment
	LDA,LDB	DCX	1					[AC0]←[[DCX]], [AC1]←[[DCX]], and [DCX]←[DCX]-1 Load AC0 or AC1 using implied addressing with auto decrement
	SMA,SMB	DCX	1					[[DCX]]←[AC0], [[DCX]]←[AC1] Store AC0 or AC1 in memory using implied addressing
	SNA,SNB	DCX	1					[[DCX]]←[AC0], [[DCX]]←[AC1], and [DCX]←[DCX] · 1 Store AC0 or AC1 in memory using implied addressing with auto increment
	SDA,SDB	DCX	1					[[DCX]]←[AC0], [[DCX]]←[AC1], and [DCX]←[DCX]-1 Store AC0 or AC1 in memory using implied addressing with auto decrement
	LSA,LSB	DCX,ADDR	2					[AC0]←[[DCX]], [AC1]←[[DCX]] and [DCX]←[DCX] · 1 plus skip Load AC0 or AC1 using implied addressing with auto increment and skip
	SSA,SSB	DCX,ADDR	2					[[DCX]]←[AC0], [[DCX]]←[AC1] and [DCX]←[DCX] · 1 plus skip Store AC0 or AC1 in memory using implied addressing with auto increment and skip

Tabella 7-1. (Continua)

TIPO	MNEMONICO	OPERANDO (I)	BYTE	STATI				OPERAZIONE EFFETTUATA
				C	O	Z	S	
Riferimento in memoria secondario	ABA,ABB	ADDR	3	X	X	X	X	[AC0]←[AC0] + [ADDR] + [C], [AC1]←[AC1] + [ADDR] + [C] Add binary with carry to A0 or A1 using direct addressing
	ABA,ABB	DCX	1	X	X	X	X	[AC0]←[AC0] + [[DCX]] + [C], [AC1]←[AC1] + [[DCX]] + [C] Add binary with carry to A0 or A1 using implied addressing
	ADA,ADB	ADDR	3	X	X	X	X	[AC0]←[AC0] + [ADDR] + [C], [AC1]←[AC1] + [ADDR] + [C] Add decimal with carry to A0 or A1 using direct addressing
	ADA,ADB	DCX	1	X	X		X	[AC0]←[AC0] + [[DCX]] + [C], [AC1]←[AC1] + [[DCX]] + [C] Add decimal with carry to A0 or A1 using implied addressing
	DSA,DSB	ADDR	3	X	X	X	X	[AC0]←[AC0] - [ADDR] - [C], [AC1]←[AC1] - [ADDR] - [C] Subtract decimal with borrow from A0 or A1 using direct addressing
	DSA,DSB	DCX	1	X	X	X	X	[AC0]←[AC0] - [[DCX]] - [C], [AC1]←[AC1] - [[DCX]] - [C] Subtract decimal with borrow from A0 or A1 using implied addressing
	ANA,ANB	ADDR	3			X		[AC0]←[AC0]Λ[ADDR], [AC1]←[AC1]Λ[ADDR] AND with A0 or A1 using direct addressing
	ANA,ANB	DCX	1			X		[AC0]←[AC0]Λ[[DCX]], [AC1]←[AC1]Λ[[DCX]] AND with A0 or A1 using implied addressing
	ORA,ORB	ADDR	3			X		[AC0]←[AC0]V[ADDR], [AC1]←[AC1]V[ADDR] OR with A0 or A1 using direct addressing
	ORA,ORB	DCX	1			X		[AC0]←[AC0] V [[DCX]], [AC1]←[AC1] V [[DCX]] OR with A0 or A1 using implied addressing
	XRA,XRB	ADDR	3			X		[AC0]←[AC0]∀[ADDR], [AC1]←[AC1]∀[ADDR] Exclusive OR with A0 or A1 using direct addressing
	XRA,XRB	DCX	1			X		[AC0]←[AC0]∀[[DCX]], [AC1]←[AC1]∀[[DCX]] Exclusive OR with A0 or A1 using implied addressing
	CMA,CMB	DCX	1	X	X	X	X	Compare memory with A0 or A1 using implied addressing
Immediato	LIM	R,DATA	2					[DST]←DATA Load immediate into R (R = A0 or A1)
	LIM	R,DATA	3					[DST]←DATA Load immediate into R (R = DC0, DC1, DC2, SP)

Tabella 7-1. (Continua)

TIPO	MNEMONICO	OPERANDO (I)	BYTE	STATI C	STATI O	STATI Z	STATI S	OPERAZIONE EFFETTUATA
Operazione immediata	AIA, AIB	DATA	2	X	X	X	X	[AC0]←[AC0] · DATA · [C], [AC1]←[AC1] · DATA · [C] Add binary immediate to A0 or A1
	NIA, NIB	DATA	2			X		[AC0]←[AC0]Λ DATA, [AC1]←[AC1]Λ DATA AND immediate with A0 or A1
	OIA, OIB	DATA	2			X		[AC0]←[AC0] V DATA, [AC1]←[AC1] V DATA OR immediate with A0 or A1
	CIA, CIB	DATA	2	X	X	X	X	Compare immediate with A0 or A1
Salto	JMP	ADDR	3					[PC]←ADDR Jump to instruction with label ADDR
	JSR	ADDR	3					[[SP]]←[PC], [PC]←ADDR, [SP]←[SP] · 1 Jump to subroutine starting at ADDR
Salto condizionato	BE, BZ	DISP	2					If [Z] = 1, [PC]←[PC] · DISP Branch on Z = 1
	BNZ	DISP	2					If [Z] = 0, [PC]←[PC] · DISP Branch on Z = 0
	BGE, BC	DISP	2					If [C] = 1, [PC]←[PC] · DISP Branch on C = 1
	BNC, BL	DISP	2					If [C] = 0, [PC]←[PC] · DISP Branch on C = 0
	BO	DISP	2					If [O] = 1, [PC]←[PC] · DISP Branch on O = 1
	BNO	DISP	2					If [O] = 0, [PC]←[PC] · DISP Branch on O = 0
	BP	DISP	2					If [S] = 0, [PC]←[PC] · DISP Branch on S = 0
	BN	DISP	2					If [S] = 1, [PC]←[PC] · DISP Branch on S = 1

Tabella 7-1. (Continua)

TIPO	MNEMONICO	OPERANDO (I)	BYTE	STATI C	STATI O	STATI Z	STATI S	OPERAZIONE EFFETTUATA
Trasferimento tra registri	MOV	SRC,DST	1					[DST]←[SRC] Move contents of SRC to DST SRC=A0, A or SP. DST=A1, DC0, DC1, DC2 or PC.
	X	SRC,DST	1					[DST]←→[SRC] Exchange SRC and DST contents
Operazioni tra registri	AB		1	X	X	X	X	[AC0]←[AC0] + [AC1] + [C] Add binary A1 to A0
	AD		1	X	X	X	X	[AC0]←[AC0] + [AC1] + [C] Add decimal A1 to A0
	SD		1	X	X	X	X	[AC0]←[AC0] - [AC1] - [C] Subtract decimal A1 from A0
	AND		1			X		[AC0]←[AC0] Λ [AC1] AND A1 with A0
	OR		1			X		[AC0]←[AC0] V [AC1] OR A1 with A0
	XOR		1			X		[AC0]←[AC0] ∀ [AC1] Exclusive OR A1 with A0
	CMP		1			X		Compare A1 with A0
	DAD	SRC,DST	1	X	X	X	X	[DST]←[DST] + [SRC] + [C] Add binary SRC to DST SRC=A0 or A. DST=DC0, DC1, DC2 or SP.
Operazione per singolo registro	SHRA,SHRB		1					Shift A0 or A1 right, simple
	SHLA,SHLB		1					Shift A0 or A1 left simple
	RORA,RORB		1					Rotate A0 or A1 right, simple
	ROLA,ROLB		1					Rotate A0 or A1 left, simple
	SRCA,SRCB		1	X				Shift A0 or A1 right through carry
	SLCA,SLCB		1	X				Shift A0 or A1 left through carry
	SRBA,SRBB		1	X				Shift A0 or A1 right with branch carry
	SLBA,SLBB		1	X				Shift A0 or A1 left with branch carry

Tabella 7-1. (Continua)

TIPO	MNEMONICO	OPERANDO (I)	BYTE	STATI				
				C	O	Z	S	
Operazione per singolo registro (continua)	SRAA,SRAB		1	X	X			Shift A0 or A1 right arithmetic
	SLAA,SLAB		1	X	X			Shift A0 or A1 left arithmetic
	SR4A,SR4B		1					Shift A0 or A1 right 4 bits
	SL4A,SL4B		1					Shift A0 or A1 left 4 bits
	SR4		1					Shift A0 and A1 right 4 bits
	SL4		1					Shift A0 and A1 left 4 bits
	INC	R	1				X	[R]←[R] + 1 Increment Register (R=A0, A1, A, DC0, DC1, DC2)
Stack	PUSH	R	1					[[SP]]←[R], [SP]←[SP] + 1 Push Register R contents onto Stack R=A0, A1, DC0, DC1, DC2, SP, PC
	POP	R	1					[R]←[[SP]], [SP]←[SP] -1 Pop top of Stack to Register R
	RET		1					[PC]←[[SP]], [SP]←[SP] -1 Same as POP PC
Passivo di param.	SPP	R	1					[R]←[[[SP]]], [[SP]]←[[SP]] + 1 Pass parameter to Register R R=A0 A1, DC0, DC1 or DC2
Interrupt	DI		1					Disable interrupts
	EI		1					Enable interrupts
	RTI		1					[SW]←[[SP]], [PC]←[[SP]] Return from interrupt
Stato	SET	I	1	X	X	X	X	[I]←1 Set Status I to 1 (I = C, O, Z, S)
	RFT	I	1	X	X	X	X	[I]←0 Reset status I to 0
	HALT		1					Halt

APPENDICE A
CODICE CARATTERE STANDARD

Rappresentazione esadecimale	ASCII (7 bit)	EBCDIC (8 bit)	Rappresentazione esadecimale	ASCII (7 bit)	EBCDIC (8 bit)
0			3F	?	
1			40	@	blank
2			41	A	
3			42	B	
4			43	C	
5			44	D	
6			45	E	
7			46	F	
8			47	G	
9			48	H	
A			49	I	
B			4A	J	|
C			4B	K	.
D			4C	L	<
E			4D	M	(
F			4E	N	+
10			4F	O	!
11			50	P	&
12			51	Q	
13			52	R	
14			53	S	
15			54	T	
16			55	U	
17			56	V	
18			57	W	
19			58	X	
1A			59	Y	
1B			5A	Z	|
1C			5B	[	$
1D			5C	\	*
1E			5D	]	)
1F			5E		;
20	blank		5F		^
21	!		60		
22	"		61	a	
23	#		62	b	
24	$		63	c	
25	%		64	d	
26	&		65	e	
27	'		66	f	
28	(		67	g	
29	)		68	h	
2A	*		69	i	
2B	+		6A	j	
2C	,		6B	k	,
2D	-		6C	l	%
2E	.		6D	m	-
2F	/		6E	n	>
30	0		6F	o	?
31	1		70	p	
32	2		71	q	
33	3		72	r	
34	4		73	s	
35	5		74	t	
36	6		75	u	
37	7		76	v	
38	8		77	w	
39	9		78	x	
3A	:		79	y	
3B	;		7A	z	
3C	<		7B		#
3D	=		7C		@
3E	>		7D		'

APPENDICE A (continua)

Rappresentazione esadecimale	ASCII (7 bit)	EBCDIC (8 bit)
7E		=
7F		"
80		
81		a
82		b
83		c
84		d
85		e
86		f
87		g
88		h
89		i
8A		
8B		
8C		
8D		
8E		
8F		
90		
91		j
92		k
93		l
94		m
95		n
96		o
97		p
98		q
99		r
9A		
9B		
9C		
9D		
9E		
9F		
A0		
A1		
A2		s
A3		t
A4		u
A5		v
A6		w
A7		x
A8		y
A9		z
AA		
AB		
AC		
AD		
AE		
AF		
B0		
B1		
B2		
B3		
B4		
B5		
B6		
B7		
B8		
B9		
BA		
BB		
BC		

Rappresentazione esadecimale	ASCII (7 bit)	EBCDIC (8 bit)
BF		
C0		
C1		A
C2		B
C3		C
C4		D
C5		E
C6		F
C7		G
C8		H
C9		I
CA		
CB		
CC		
CD		
CE		
CF		
D0		
D1		J
D2		K
D3		L
D4		M
D5		N
D6		O
D7		P
D8		Q
D9		R
DA		
DB		
DC		
DD		
DE		
DF		
E0		
E1		
E2		S
E3		T
E4		U
E5		V
E6		W
E7		X
E8		Y
E9		Z
EA		
EB		
EC		
ED		
EE		
EF		
F0		0
F1		1
F2		2
F3		3
F4		4
F5		5
F6		6
F7		7
F8		8
F9		9
FA		
FB		
FC		
FD		

L. 18.000

26

INTRODUZIONE AI
MICROCOMPUTER

VOLUME 1 Il libro
dei concetti fondamentali